Apple III

Pascal Technical Reference Manual

Acknowledgements

The Apple III Pascal system is based on UCSD Pascal. "UCSD PASCAL" is a trademark of the Regents of the University of California. Use thereof in conjunction with any goods or services is authorized by specific license only and is an indication that the associated product or service has met quality assurance standards prescribed by the University. Any unauthorized use thereof is contrary to the laws of the State of California.

Contents

3 P-Machine 35

4 Assembly-Language Programming 53

5 The P-Machine Instruction Set 63

Figures and Tables

2 Codefiles 5

3 The P-Machine 35

Preface

The *Apple III Pascal Technical Reference Manual* is a technical reference for more advanced users of the Apple III Pascal system. It describes the architecture and operation of the P-machine, operating system, and I/O of the Apple III Pascal system. Before you use the information it contains, you should be familiar with these manuals:

Apple III SOS Reference Manual

Apple III Pascal: Introduction, Filer, and Editor

Apple III Pascal Programmer's Manual, Volumes 1 and 2

Apple III Pascal Program Preparation Tools

Many of the concepts explained in this volume are intimately interrelated. You should first briefly read the entire book and gain an appreciation of how the concepts are interrelated before attempting to understand any specific concept in detail. Here is a brief description of the contents:

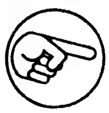
- Chapter 1 is an overview of the Apple III Pascal system.
- Chapter 2 describes the structure and format of codefiles on disk.
- Chapter 3 describes the structure and format of code in memory, and the operation of the P-machine.
- Chapter 4 details the use of assembly-language procedures and functions.
- Chapter 5 describes the P-machine instruction set.

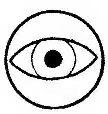
- Chapter 6 contains useful assembly-language and Pascal programming techniques and hints.

A glossary and an index are also included. Any item that appears in the Glossary is shown in boldface type at its first occurrence in the text of this manual.

You should be familiar with the hexadecimal numbering system. Hexadecimal numbers in the text and tables of this manual are preceded by a dollar sign ($). In the text or in a table or illustration, any number that is not preceded by a dollar sign is a decimal number.

Two special symbols are used in this manual to draw your attention to particular items of information:


The pointing hand indicates something particularly interesting or useful.


The eye indicates points you need to be cautious about.

1

Overview

1
Overview

The Pascal Environment

The Apple III Pascal system is a version of the UCSD Pascal system, a psuedo-machine-based implementation of Pascal. This means that the Compiler converts Pascal program text into compact **pseudo-code** or **P-code** to be executed by the **pseudo-machine** or **P-machine**. The P-machine is implemented by the Pascal interpreter—a program written in the **native code** of the Apple III's 6502 microprocessor. Every host computer operating under a version of UCSD Pascal has such an interpreter that makes the host computer appear, from the viewpoint of a program being executed, to be a P-machine. The interpreter is contained in the SOS.INTERP file on the PASCAL1 disk.

The Pascal operating system and various utility programs are also written in Pascal and run on the same interpreter. The Pascal system runs "on top of" SOS, the Apple III operating system. (See the *Apple III SOS Reference Manual* for a detailed explanation of this relationship.)

The Pascal Compiler, Assembler, and **Linker** together produce completed **codefiles** of Pascal programs. Pascal codefiles are stored on external storage media, such as disks. The structure of codefiles is explained in Chapter 2. When a Pascal program is to be executed, the interpreter loads the code of the user program main segment of the codefile into memory, and then begins executing the program code, one instruction at a time. As the interpreter finds that additional segments of the disk codefile are needed in memory for execution of the program, it loads the necessary segments. The structure and execution of code in memory is described in Chapter 3. Pascal programs can

contain assembly-language procedures and functions; these are discussed in Chapter 4. The P-machine instruction set is described in Chapter 5. A group of useful Pascal and assembly-language programming techniques are discussed in Chapter 6.

2

Codefiles

2
Codefiles

Codefiles may be (1) **linked files** composed of **segments** ready for execution, (2) **library files** with units which may be used by programs in other codefiles, or (3) **unlinked files** created by the Compiler or Assembler. A typical disk codefile resulting from the compilation of a program is diagrammed in Figure 2-1.

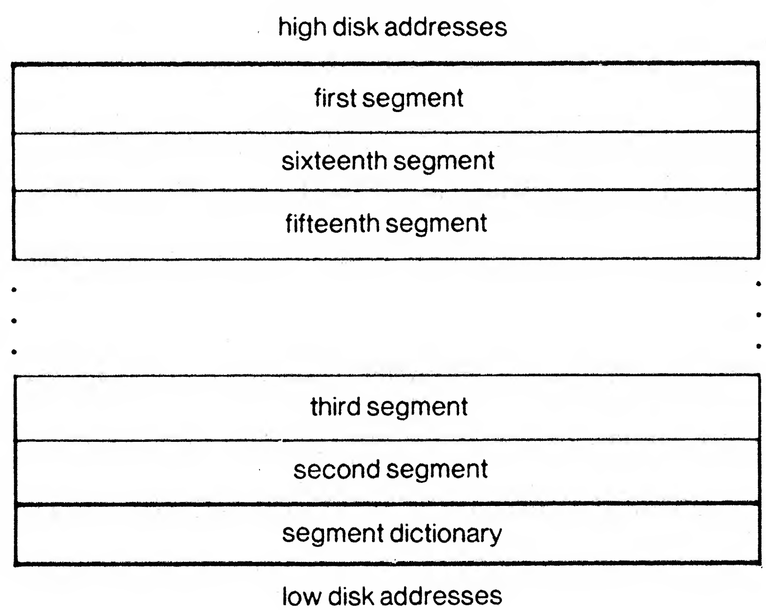


Figure 2-1. A Typical Codefile on Disk

All codefiles (linked and unlinked) consist of a **segment dictionary** in block 0 of the codefile, followed by a sequence of one to 16 code segments. The **host program** is compiled into one code segment, and each **SEGMENT procedure**, **SEGMENT function**, and **unit** is translated into another code segment. The ordering of code segments in the codefile (from low disk address to high disk address) is determined by the order in which the Compiler encounters the executable code of each SEGMENT procedure, SEGMENT function, and unit when compiling a program. This order may be changed by the **Librarian**.

Each segment begins on a boundary between disk blocks (a **block** is 512 contiguous **bytes**). Each segment may occupy up to 64 blocks.

Segments

A **segment** is either a **code segment** or a **data segment**. Program code is stored in code segments. Every program consists of at least one code segment, and some programs consist of many code segments. A code segment may contain either P-code, native 6502 code, or a combination of both. Code segments may have three parts: **interface text**, actual P-code and/or native code, and **Linker information** (Figure 2-2). These parts appear in this order on the disk, although not all types of code segments have all three parts. For example, interface text is present only in the code segments of units. Code segments may be either linked or unlinked.

Data segments are areas of memory that are set aside at execution time as storage space for the local data of **intrinsic units**. In a disk codefile, data segments have only an entry in the segment dictionary: they do not occupy any blocks on the disk since they have no code part, interface text, or Linker information associated with them.

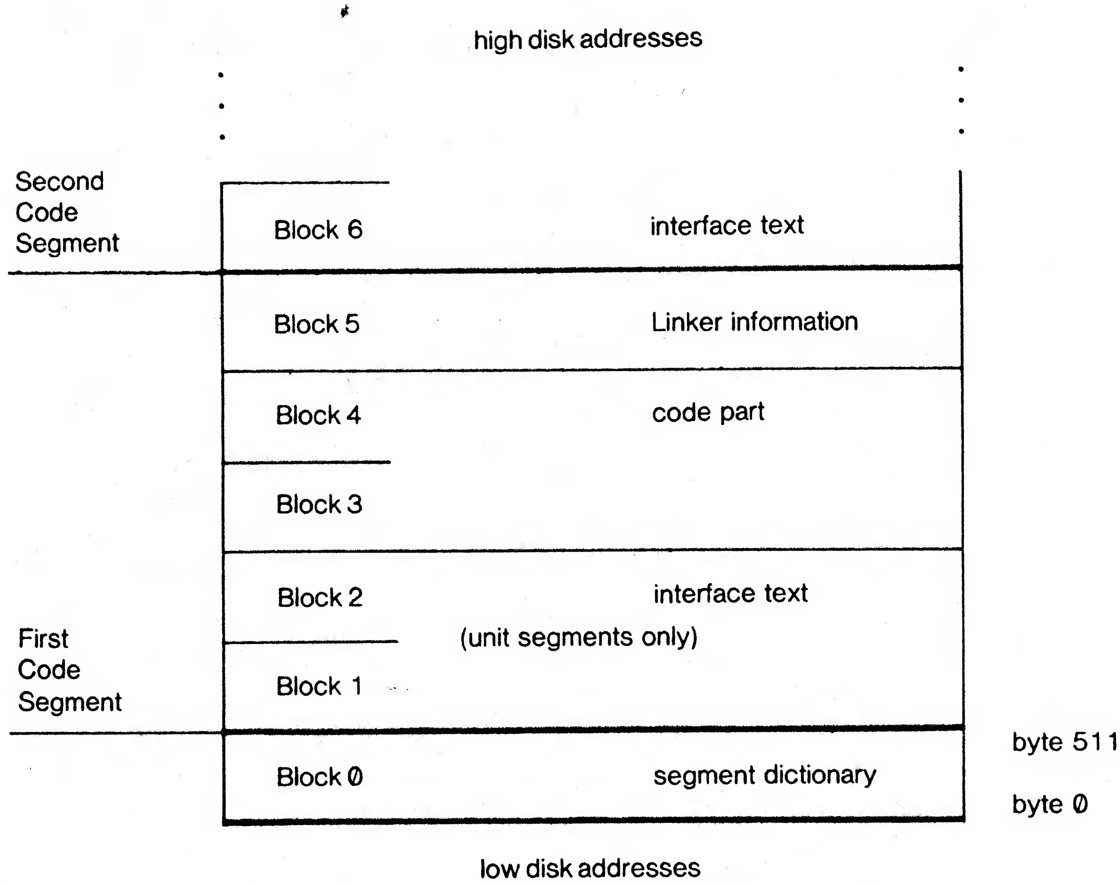


Figure 2-2. A Typical Codefile

Note that Figure 2-2 is not meant to imply that all code segments are five blocks long; the code part of a segment can contain up to 64 blocks.

Whenever a complete program codefile is produced by the Compiler (and Assembler and Linker, if necessary), the following occur:

- The user program or unit results in one code segment in the codefile. This includes the user program's non-SEGMENT procedures and functions (MULT2 and STOR in Figure 2-3), and the user program body itself (MAIN in Figure 2-3).
- Each Pascal SEGMENT procedure or function results in a another code segment in the codefile (BYFOUR and DIVID below).
- Each regular unit that the program USES is linked with the codefile and results in a code segment in the codefile (REGUNIT below). Each intrinsic unit that the program USES does not produce additional code

segments in the program's codefile. Intrinsic units are held as segments in program libraries, shared libraries, and the **SYSTEM.LIBRARY file**, and accessed by the program at **execution time** (MAINLIBIU below).

Source text files

```
PROGRAM MAIN;
    USES MAINLIBIU, REGUNIT;
    SEGMENT FUNCTION DIVID;
    BEGIN
      .
      .
    END;
    SEGMENT PROCEDURE BYFOUR;
    BEGIN
      .
      .
    END;
    FUNCTION MULT2;
    BEGIN
    .
    .
    END;
    PROCEDURE STOR;
    BEGIN
    .
    .
    END;
BEGIN
  .
  .
END.

UNIT REGUNIT;
BEGIN
  .
  .
END.

UNIT MAINLIBIU; INTRINSIC
    CODE 40 DATA 41;
BEGIN
  .
  .
END.
```

Segments in codefile after linking

REGUNIT code segment
MAIN "outer" code segment MULT2 function STOR procedure
BYFOUR code segment
DIVID code segment

Segment in Library

MAINLIBIU code segment

Figure 2-3. Correlation Between Programs and Segments in Codefiles

Segments are not nested in codefiles as they are in programs. Instead every segment is a separate contiguous area of code and does not contain any other segments. For example, if a SEGMENT procedure contains another SEGMENT procedure, the compiled result comprises two distinct code segments, even though they are nested lexically in the source program.

Segmenting a program does not change the computation it performs. When a SEGMENT procedure, SEGMENT function, or intrinsic unit is called during the execution of a program, the interpreter checks to see if that segment is already in memory due to a previous (and still active) invocation of the segment. If it is, control is transferred and execution proceeds; if not, the appropriate code segment is loaded into memory from the disk codefile before the transfer of control takes place. When no more active invocations of a segment exist, its code is removed from memory.

The following sections describe the portions of a code segment in greater detail. First the segment dictionary is described. Then the parts of a code segment are presented in the order in which they may occur in a codefile: the interface text, the code part, and finally the Linker information. The code part description is divided into sections describing the similarities and differences between the code parts of P-code and assembly-language procedures.

Segment Dictionaries

Every codefile (including library files) has a **segment dictionary** in block Ø that contains information needed by the Pascal system to load and execute the segments in that codefile. A segment dictionary has 16 **slots**, each of which either contains information about one segment, or is empty. Each non-empty slot includes the segment's name, kind, size (in bytes), and location in the codefile. The location of a code segment is given as the block number of the first block in the code part (blocks in a codefile are numbered sequentially from zero, with block Ø as the segment dictionary). The location of a data segment is given as zero.

The information that describes each segment is contained in five different arrays within the segment dictionary. All information describing a segment is retrieved by selecting corresponding elements from each of these arrays.

Since a segment dictionary has 16 slots, numbered 0 through 15, one codefile can contain at most 16 segments. Intrinsic units used by a program do not require entries in the segment dictionary of the program's codefile, because intrinsic unit code segments are in a library file, and appear in the library file's segment dictionary. Therefore, a program can have a maximum of 16 segments, not counting segments from intrinsic units. Counting intrinsic units, the maximum number of segments is limited by the total number of segment numbers in the system (64). However, the system reserves eleven segment numbers (0, 2 through 6, and 59 through 63) for its own use. The remaining 53 segments may appear in a program codefile, a program library file, a SYSTEM.LIBRARY file, or library files specified in a **library name file**. Each of these codefiles can contain a maximum of 16 segments.

The following Pascal-like record **declaration** represents a segment dictionary:

```
RECORD

  DISKINFO:  ARRAY[0..15] OF
    RECORD
      CODEADDR:  INTEGER;  {location of code part}
      CODELENG:  INTEGER   {length of code part}
    END;

  SEGNAME:  ARRAY[0..15] OF PACKED ARRAY[0..7] OF
            CHAR;              {segment name}

  SEGKIND:  ARRAY [0..15] OF {type of segment}
        (LINKED,               {fully executable segment}
         HOSTSEG,              {user program code segment}
         SEGPROC,              {unused}
         UNITSEG,              {compiled regular unit}
         SEPRTSEG,             {separate procedures and
                               functions}
         UNLINKED-INTRINS,     {unlinked intrinsic unit}
         LINKED-INTRINS,       {linked intrinsic unit}
         DATASEG);             {data segment}

  TEXTADDR:  ARRAY[0..15] OF INTEGER;  {address of the
             first block of interface text, if any}

  SEGINFO:  PACKED ARRAY[0..15] OF PACKED RECORD

      SEGNUM:  0..255; {segment number}
      MTYPE:  0..15;   {machine type}
      UNUSED:  0..1;   {unused}
      VERSION:  0..7   {version number}

      END;
```

```
INTRINS-SEGS:  SET OF Ø..63;  {intrinsic segment
                               numbers needed for
                               execution}

INT-NAM-CHECKSUM:  PACKED ARRAY [Ø..63] OF
Ø..255;                         {checksum}

FILLER:  PACKED ARRAY [1..72] OF Ø.255; {72 unused
                               bytes filled with
                               zeros}

COMMENT:  PACKED ARRAY [Ø..79] OF CHAR {comment}

END;
```

The following diagram (Figure 2-4) indicates the structure of a segment dictionary:

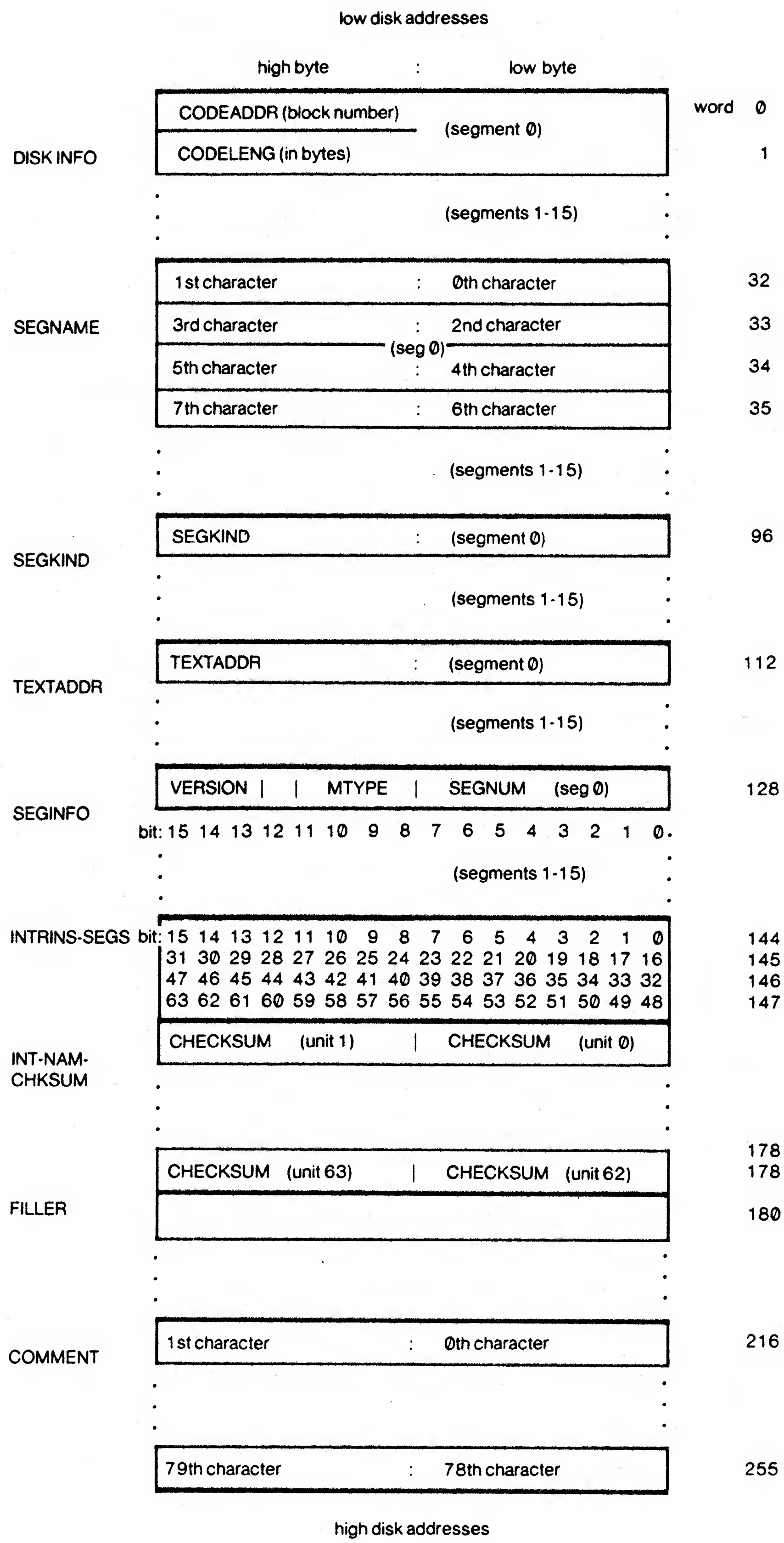


Figure 2-4. A Segment Dictionary

Note that the diagram in Figure 2-4 shows lower addresses at the top (in contrast to others in this manual) to match the structure of the Pascal-like segment dictionary declaration.

A segment dictionary is composed of five 16-element arrays (one element for each segment slot in the segment dictionary of a codefile), information on the intrinsic segments used by the codefile, and a comment.

Each element in the DISKINFO array consists of two **words** that describe the length and location of the segment within the codefile. For code segments, the CODEADDR field contains the block number of the start of the code part, and the CODELENG field contains the number of bytes in the code part of the segment. For data segments, the CODEADDR field is zero, and the CODELENG field contains the number of bytes to be allocated for data at execution time (the length of the data segment). Unused slots have their CODEADDR and CODELENG fields set to zero (CODELENG=Ø defines an empty slot).

Each element of the SEGNAME array is an eight-character array which contains the first eight characters of the user program, unit, SEGMENT procedure, SEGMENT function, or assembly-language procedure name that was translated into the corresponding segment. If the name is shorter than eight characters, it is padded on the right by spaces; if the name is longer than eight characters, it is truncated to the first eight characters. Unused segment slots have SEGNAME fields filled with eight ASCII space characters.

The SEGKIND array describes the type of segment. The possible values are:

Ø: LINKED. A fully-executable segment. Either all references to regular or intrinsic units have been resolved by the Linker, or none were present.

1: HOSTSEG. The main segment of a user program with unresolved external references.

2: SEGPROC. A SEGMENT procedure or function. This type is currently not used.

3: UNITSEG. A compiled regular (as opposed to intrinsic) unit.

4: SEPRTSEG. A separately assembled (set of) procedures or functions, including EXTERNAL functions and procedures, and mixed segments of linked Pascal and assembly-language code. Assembly-language codefiles are always of this type.

5: UNLINKED-INTRINS. An intrinsic unit containing unresolved calls to assembly-language procedures or functions.

6: LINKED-INTRINS. An intrinsic unit properly linked with its called procedures and functions.

7: DATASEG. A data segment of an intrinsic unit. The segment dictionary entry specifies the amount of data space (in bytes) to allocate.

The TEXTADDR array of integers contains pointers to the block number of the start of the interface text of each regular or intrinsic unit. The last block number of the interface text can be calculated by subtracting 1 from the value in the corresponding CODEADDR field. Interface text is described in detail below. Only unit segments have interface text; the TEXTADDR field is zero for all other types of segments.

The SEGINFO array contains one word of additional information about each segment. Each word is composed of four fields:

Bits Ø through 7 (the low-order byte) of each word specify the **segment number** (SEGNUM). This is the position the code segment will occupy in the **segment table** at execution time. The segment table is 64 entries long, hence valid numbers for the SEGNUM field are Ø..63. (Segment tables are described in Chapter 3).

Bits 8 through 11 of the second byte in the SEGINFO word specify the **machine type** (MTYPE) of the code in the segment. The machine types are:

Ø: Unidentified code, perhaps from another Compiler.

1: P-code, most significant byte first.

2: P-code, least significant byte first (a stream of packed ASCII characters fills the low byte of a word first, then the high byte). This is the kind of P-code used by the Apple III.

3 through 9: Assembled native code, produced from assembly-language text. Machine type 7 identifies native code for the Apple III's 65Ø2 microprocessor.

Bit 12 of the SEGINFO word is unused.

Bits 13 through 15 of the SEGINFO word contain the version number of the system. The current version number is 3, indicating Apple III Pascal.

The SEGINFO array is the last of the five arrays that contain 16 elements, one element for each slot. The remainder of the segment dictionary contains information pertinent to the execution of the entire codefile.

The INTRINS-SEGS field consists of four words (64 bits). These four words specify which intrinsic units are needed to execute the codefile. Each intrinsic unit in a program library file, SYSTEM.LIBRARY file, and library file specified in a library name file, is identified by a segment number (or two segment numbers if the intrinsic unit has both a code and data segment). Each one of the 64 bits in these words corresponds to one of the 64 possible intrinsic segment numbers. If the nth bit is set, the codefile needs the intrinsic unit whose segment number is n in order to execute. Bits corresponding to the segment numbers of unused intrinsic units are zeroed.

Some intrinsic units are part of the Pascal operating system. While their use is indicated by set bits in the INTRINS-SEGS field, they are not loaded from either the SYSTEM.LIBRARY or the **program library**, but are present at execution time. These special segments are numbered 59 through 63.

The INT-NAM-CHECKSUM array contains 64 fields of 8-bit checksums of the names of the intrinsic units needed to execute the codefile. Each field corresponds to one of the 64 possible intrinsic segment numbers. These checksums are used by the Pascal operating system to ensure that two differently-named segments with identical segment numbers are not confused. The checksum is calculated by shifting the characters of the unit name to uppercase and summing the resulting ASCII values of the characters of the unit name MOD 256. The name is padded with spaces on the right if it is

shorter than eight characters; it is truncated to eight characters if it is longer than eight characters. Padding spaces are included in the checksums. Words corresponding to the segment numbers of unused intrinsic segments are filled with blanks.

The FILLER array contains 72 unused bytes.

The COMMENT array contains text provided by a **Compiler COMMENT option** or when the Librarian is used. It starts in word 216 of the segment dictionary.

Segment Numbers

At execution time, every segment has a segment number from 0 to 63, and no two segments in the program can have the same number. Segment numbers are assigned as follows (Figure 2-5):

- the user program itself is segment 1.
- the segments used by the Pascal operating system are 0, 2 through 6, and 59 through 63. These numbers are never assigned to segments of the user program.
- the segment number of an intrinsic unit segment is determined by the unit's heading when the unit is compiled. (These numbers can be found by using the **LIBMAP utility program** to examine the segment dictionary of the library to which the unit belongs.
- the segment numbers of **regular unit segments** and of SEGMENT procedures and functions within the program are automatically assigned by the system as the program is compiled and linked. The segment numbers of regular units and of SEGMENT procedures and functions begin at 7 and ascend. Note that after a regular unit is linked with a program, it may not have the same segment number that was shown for it in the library's segment dictionary (when examined with the LIBMAP utility), because the Linker may reassign segment numbers of regular units.

Segment Number	Assignment
Ø	Pascal operating system
1	user program
2...6	Pascal operating system
7...29	units, SEGMENT procedures and functions
3Ø	PASCALIO unit
31	LONGINTIO unit
32...58	units, SEGMENT procedures and functions
59...63	Pascal operating system

Figure 2-5. Segment Number Assignment

Normally, only when writing an intrinsic unit do you need to specify segment numbers; this is explained in Chapter 14 of the *Apple III Pascal Programmer's Manual*. The choice must avoid the Pascal system segment numbers Ø through 6 and 59 through 63, and numbers assigned to any other intrinsic unit which may be used in the same program as the unit being written. In addition, the standard library units **PASCALIO** and **LONGINTIO** occupy segment numbers 3Ø and 31. Therefore, if you perform I/O of real numbers, long integer operations, or use the SEEK procedure, you cannot assign your own units segment numbers 3Ø and 31.

The PASCALIO and LONGINTIO standard library units are known to the Compiler and do not require a USES declaration.

Intrinsic unit segment numbers must also avoid conflict with numbers which may be assigned automatically to regular units and SEGMENT procedures and functions. In other words, use high segment numbers for intrinsic units. However, when unavoidable conflicts arise, the **NEXTSEG Compiler option** described in the *Apple III Pascal Programmer's Manual* can be used to set the segment number to another value. (Segment numbers are discussed in further detail in conjunction with the section The Segment Table in Chapter 3.)

Interface Text

Code segments of units may have interface text before their code part; host segments, SEGMENT functions and procedures, and **EXTERNAL procedures and functions** never have interface text. The interface text contains the ASCII text of the **INTERFACE** section in the source text of a unit. The construction of an interface text of a segment from its source text (by the Compiler) is shown in Figure 2-6.

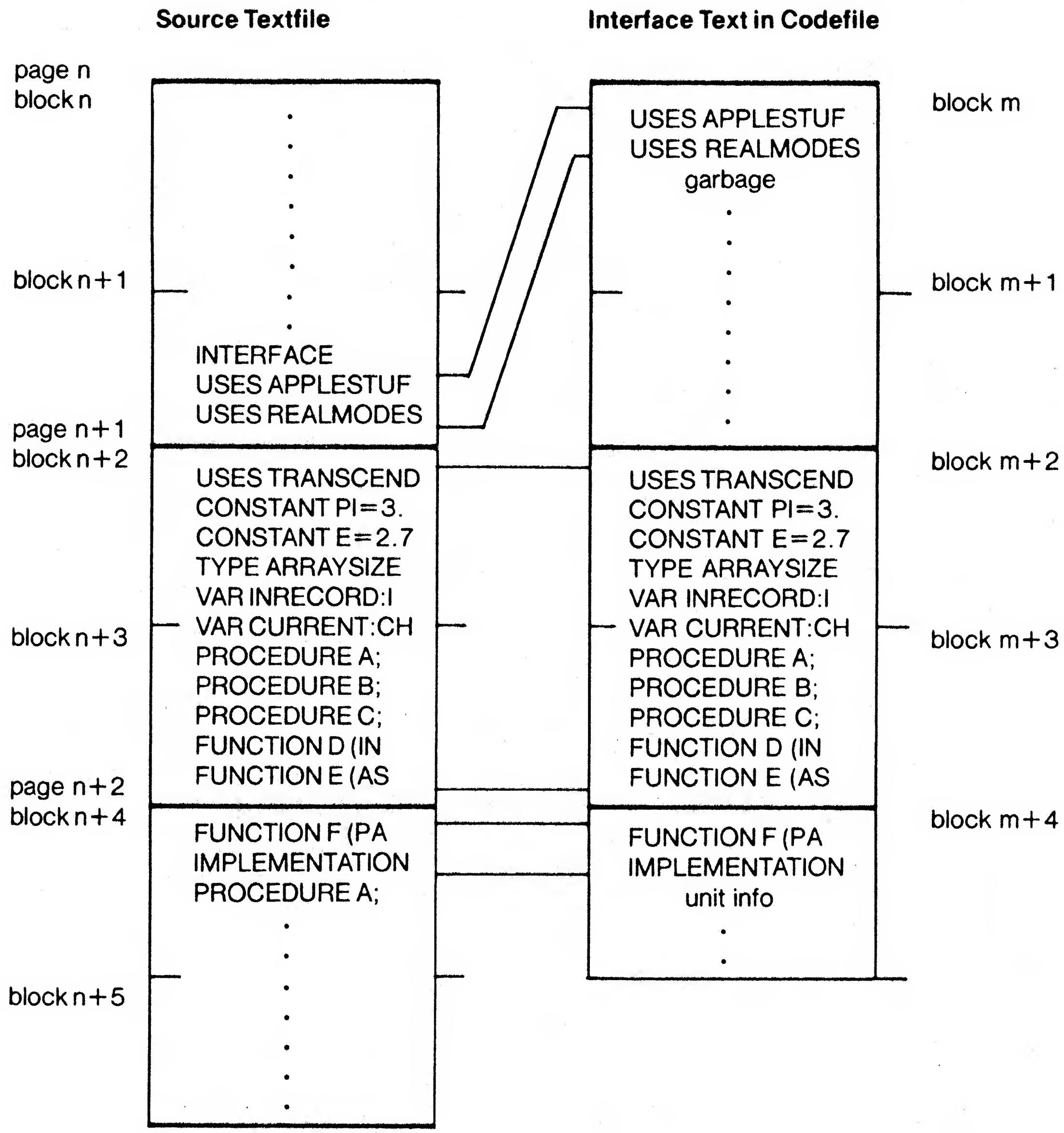


Figure 2-6. Construction of Interface Text in a Codefile

The Pascal Compiler reads source text and produces interface text in two-block **pages** (1Ø24 bytes each). Interface text always starts on a page boundary and follows all of the conventions of a **textfile**, with the exception that the last page of the interface text may be either 1 or 2 blocks long. The interface text is identical to the source text, except for the first and last pages. The information in the first page of the source text is truncated, such that the first character in the output page is the character following the INTERFACE keyword in the original source text ("U" in Figure 2-6). This may leave a considerable amount of unused space in the first page. The last page of the

source text is truncated after the **IMPLEMENTATION** keyword; it is possible that only one block of this page may be produced if the IMPLEMENTATION keyword occurs in the first block of the page. (IMPLEMENTATION is explained in the *Apple III Pascal Programmer's Manual*). Valid data in each page of a textfile end with a CR (ASCII 13) followed by at least one NULL (ASCII Ø).

The ten characters immediately following the IMPLEMENTATION keyword contain special **unit info**. All ten characters are ASCII spaces, except for an *E* in the ninth position, required by the Pascal Compiler and Librarian programs to terminate the interface text. A *P* may occur, instead of a space, in the second of the ten character positions to signify to the Pascal Compiler that the unit requires the PASCALIO standard library unit. The fourth position will be occupied by an *L* if the unit requires the LONGINTIO standard library unit. These items—IMPLEMENTATION, P, L, and E—are all considered tokens by the Compiler; thus, their order is significant, but their spacing and case are not.

Interface text is not stripped of non-printing characters or comments. The comments are not necessary for execution, but leaving the comments in the interface text can lead to more complete internal program documentation at the expense of increased codefile length. Note that the interface text of unit segments is used only during compilation. Therefore, this text can be removed from completed codefiles that will only be executed. The effect is a reduction in codefile size.

The TEXTADDR array of the segment dictionary contains pointers to the starting address of the interface text for each segment. The pointers specify block numbers, relative to the start of the codefile. The field is zero for segments that are not unit code segments, and unit segments that do not have an interface part.

Code Parts

The code part of a code segment consists of a group of **procedures**, together with descriptive information about the procedures, called the **procedure dictionary**. A code segment may contain up to 16Ø procedures, no more than 149 of which can be P-code procedures (the remainder must be assembly-language procedures). Figure 2-7 is a diagram of the code part of a code segment. Each code part contains the code for the highest level procedure in

the segment, as well as the code for each of the non-SEGMENT procedures and functions within the segment. The code of the highest level procedure, which is generated last, appears highest in the code part.

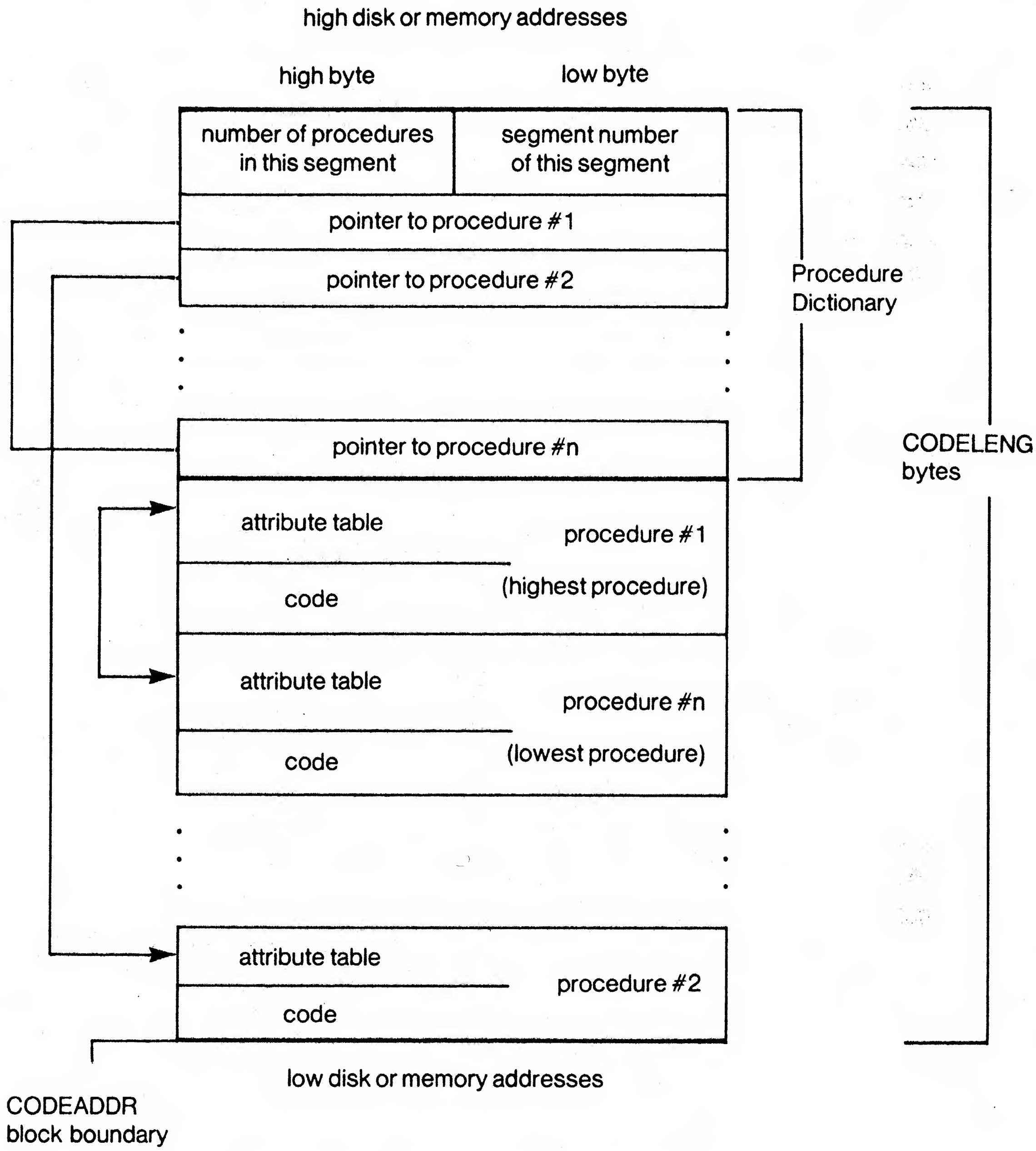


Figure 2-7. The Code Part of a Code Segment

Each procedure in a code part is assigned a **procedure number** starting at 1, for the highest level procedure or SEGMENT procedure, and ranging as high as 149. All references to a procedure are made via its segment number and procedure number. Translation from a procedure number to the location of the **procedure code** in the code segment is accomplished via the **procedure dictionary**.

Below the procedure dictionary is the code for the various procedures in the segment. The procedure dictionary grows downward toward lower disk addresses; the code part starts at the first byte of the block specified in the CODEADDR field of the segment dictionary and grows upward toward higher addresses.

Procedure Dictionaries

The position of the low-order byte of the highest word in a segment's procedure dictionary can be calculated as:

CODEADDR * 512 + CODELENG – 2

This highest word in a procedure dictionary contains the segment number in its **low-order (even) byte**, and the number of procedures in the segment in its **high-order (odd) byte**. Below this word is a sequence of words that contain self-relative pointers to the top (high address) of the code of each procedure in the segment (Figure 2-7). (A **self-relative pointer** contains the absolute distance, in bytes, between the low-order byte of the pointer and the low-order byte of the word to which it points. To find the address referred to by a self-relative pointer, subtract the pointer from the address of its location to find the byte pointed to.)

A procedure's number is an index into the procedure dictionary: the nth word in the dictionary (counting downward from higher addresses) contains a pointer to the top (high address) of the code of procedure n. As zero is not a valid procedure number, the zeroth word of the dictionary is used to store the segment number of the code segment, and the number of procedures in that code segment (as described above).

Procedures

Each procedure consists of two parts: the procedure code itself (in the lower portion of the procedure growing up toward higher addresses), and an **attribute table** of the procedure. Some procedures have a third part called the **jump table** located at the base of the attribute table. Figure 2-8 is a diagram of a typical procedure.

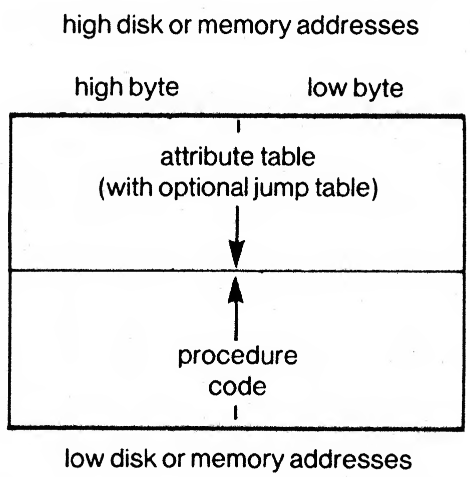


Figure 2-8. A Typical Procedure

ATTRIBUTE TABLES

The **attribute table** of a procedure provides information needed to execute the procedure. Procedure attribute tables are pointed to by entries in the procedure dictionary of each segment.

The Compiler produces P-code by compiling source text, and the Assembler produces native code by assembling assembly language. Procedures may contain solely P-code or native code, but not a mixture of both. It is possible to produce segments with procedures of both code types using the Linker. In this case the MTYPE field in the segment dictionary is set to the value for assembled native code (7), because the code for that segment is then machine-specific. The interpreter is able to determine the type of code in a particular procedure via information contained in the procedure's attribute table. The format of the attribute table for an assembly-language procedure is very different from that for a P-code procedure. These two formats are described in the following sections.

P-Code Procedure Attribute Tables

The format of a P-code procedure attribute table is illustrated in Figure 2-9.

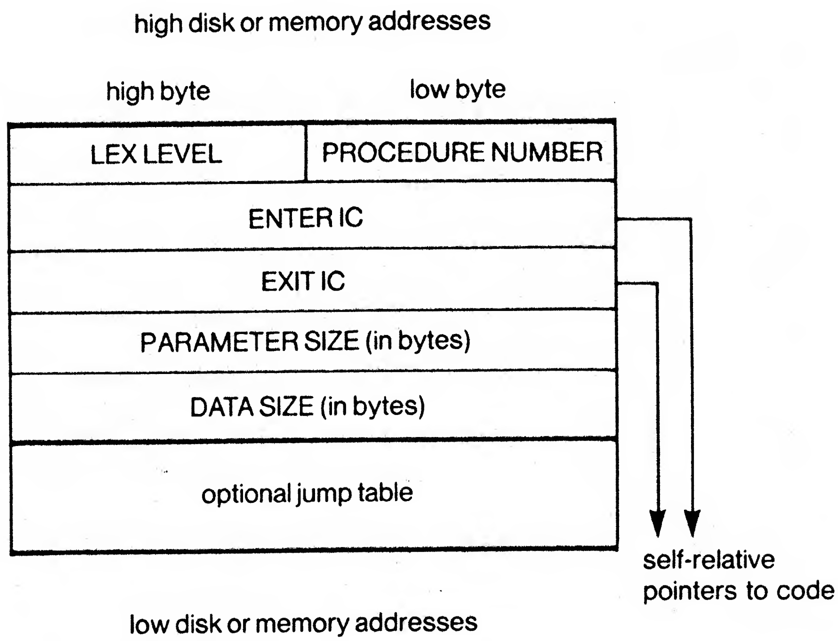


Figure 2-9. P-Code Procedure Attribute Table

The fields of a P-code procedure attribute table are:

PROCEDURE NUMBER: This field contains the **procedure number**. The PROCEDURE NUMBER field is the low-order (even) byte of the highest word in the attribute table.

LEX LEVEL: This field specifies the depth of lexical nesting of the procedure. The **lexical level** of the Pascal operating system is −1, the lexical level of a user program is 0, and that of the first nested procedure is 1, and so forth. (See the *Apple III Pascal Programmer's Manual*, Volume 2). The LEX LEVEL field is the high-order (odd) byte of the highest word in the attribute table.

ENTER IC: This field contains a self-relative pointer (again, a positive number, pointing back) to the first P-code instruction to be executed in the procedure.

EXIT IC: This field contains a self-relative pointer to the beginning of the sequence of P-code instructions that must be executed to terminate the procedure properly.

PARAMETER SIZE: This field specifies the number of bytes of parameters passed to a procedure from its calling procedure. If the procedure is a **function**, this number includes the number of bytes to be reserved for the returned value.

DATA SIZE: This field specifies the number of bytes to be reserved for local variables of the procedure.

At the base of the attribute table there may be a section called the jump table. Jump tables are used by the P-machine to determine the locations specified by jump instructions. Its entries are self-relative pointers to addresses within the procedure code. During execution, the **JTAB**, **XJTAB** psuedo-register points to the PROCEDURE NUMBER field of the attribute table of the currently executing procedure. (See Chapter 3 for an explanation of the **pseudo-registers**.)

All jump instructions include a specified jump offset (n). In the case of short forward jumps, the jump table is ignored, and execution jumps by n bytes. In the case of backward or long forward jumps, the jump offset specifies a self-relative pointer in the jump table located n bytes below the location pointed to by the JTAB register. Execution jumps to the byte address pointed to by the self-relative pointer.

Assembly-Language Procedure Attribute Tables

The format of an attribute table of an assembly-language procedure is very different from that of a P-code procedure attribute table. It is illustrated in Figure 2-10.

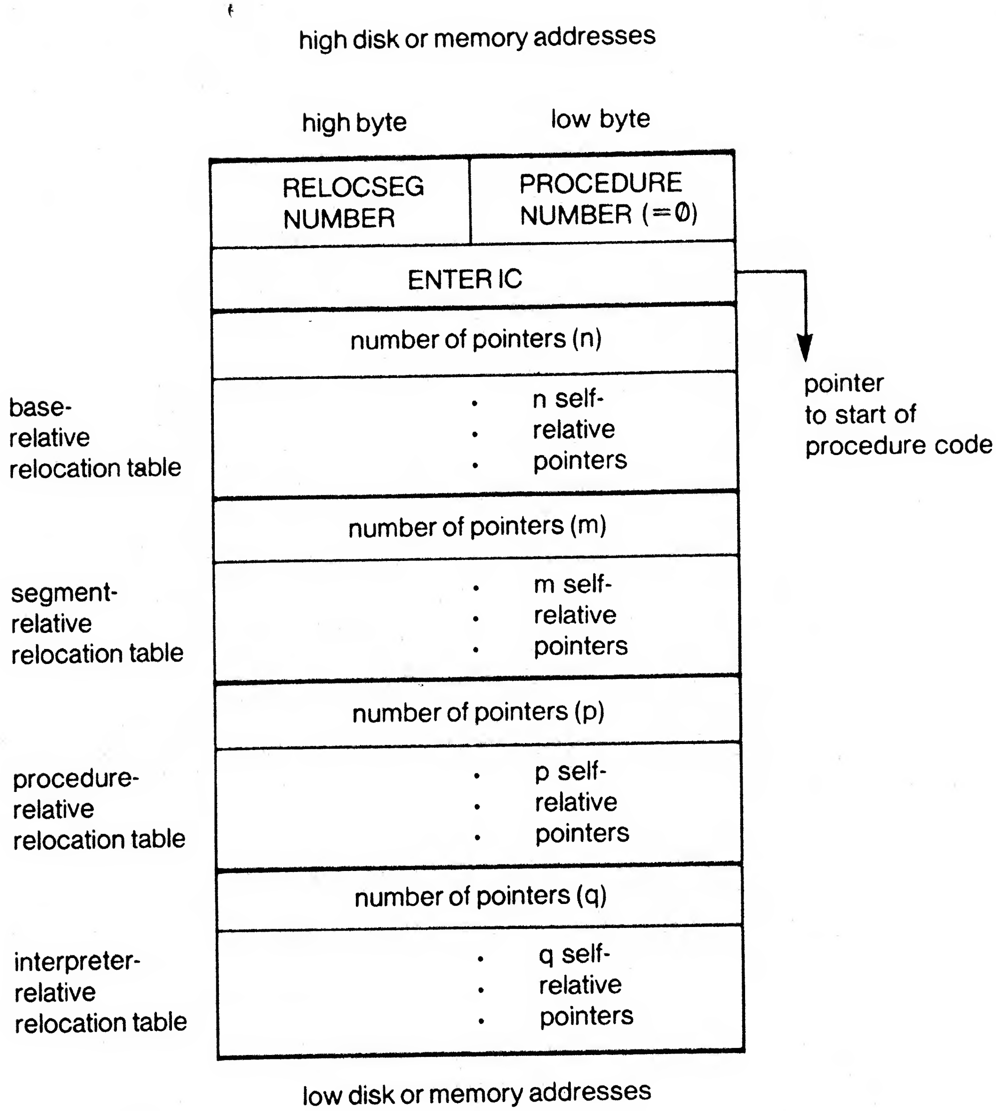


Figure 2-10. An Assembly-Language Procedure Attribute Table

The highest word in the attribute table of an assembly-language procedure always has a zero in its PROCEDURE NUMBER field. When the interpreter encounters a zero in the PROCEDURE NUMBER field as it loads the segment, it realizes it must "fix up" references in the procedure code according to information contained in the rest of the attribute table. The RELOCSEG NUMBER field contains either a zero or a positive number (the significance of which is explained below in conjunction with base-relative relocation). In the case of intrinsic units without data segments, the number placed in this field is 1. The second highest word of the attribute table is, as in P-code procedure attribute tables, the ENTER IC field—a self-relative pointer to the first

executable instruction of the procedure. Following this are four relocation tables used by the interpreter. From high address to low address, they are base-relative, segment-relative, procedure-relative, and interpreter-relative relocation tables.

Relocation Tables

A **relocation table** is a sequence of records that contain information necessary to relocate any relocatable addresses used by code within the procedure. Relocatable addresses are relocated whenever the segment containing the procedure is loaded into memory. Only native code procedures use relocatable addresses; procedures that contain P-code are completely position-independent, and no relocation list is needed.

The format of all four relocation tables is the same: the highest word of each table specifies the number of entries (possibly zero) that follow (at lower disk addresses) in the table. The remainder of each table comprises that number of one-word self-relative pointers to locations in the procedure code which must be "fixed". The locations are fixed by the addition of the appropriate relative relocation constant known to the interpreter when the segment is loaded.

Addresses pointed to by a **base-relative relocation table** are relocated relative to the address contained in the P-machine's **BASE**, **XBASE** psuedo-register if the RELOCSEG NUMBER field of the procedure's attribute table is zero. The BASE, XBASE register is a pointer to the **activation record** of the most recently invoked **base procedure** (lexical level Ø or −1). **Global** (lexical level Ø) variables are accessed by indexing from the value of the BASE register. If the RELOCSEG NUMBER field is non-zero, the relocations will be relative to the lowest address of the segment whose segment number is contained in the RELOCSEG NUMBER field. This is used by assembly procedures that are linked with intrinsic units to access the intrinsic unit's data segment. .PUBLIC and .PRIVATE are the **Assembler directives** that generate base-relative relocation fields.

Addresses pointed to by a **segment-relative relocation table** are relocated relative to the lowest address of the segment. The value of the address of the lowest byte in the segment is added to each of the addresses pointed to in the relocation table. .REF and .DEF are the Assembler directives that generate segment-relative relocation fields.

Addresses pointed to by a **procedure-relative relocation table** are relocated relative to the lowest address of the procedure. The value of the address of the lowest byte in the procedure is added to each of the addresses pointed to in the relocation table.

The interpreter-relative relocation fields point to relocatable addresses that access Pascal interpreter procedures or variables. Addresses pointed to by an **interpreter-relative relocation table** are relocated relative to a nine-word table in the interpreter. See the explanation of the .INTERP directive in the *Apple III Pascal Program Preparation Tools* manual.

Linker Information

Following the code part of a segment there may be Linker information. Linker information is the portion of a code segment that enables the Linker to resolve references of variables, identifiers, and constants between separately compiled or assembled code. Segments produced by an Assembler always have Linker information. Segments produced by the Compiler have Linker information only if they are segments with EXTERNAL procedures or units, or user programs that USE regular units.

The starting location of Linker information is not included in the segment dictionary as was the case with the starting location of the interface text and code parts; it must be inferred. Linker information starts on the **block boundary** following the last block of code for a segment, and grows toward higher addresses. The block number of the first record of Linker information can be calculated as:

CODEADDR + ((CODELENG + 511) DIV 512)

where CODEADDR and CODELENG are the values of fields in the segment dictionary.

Linker information is stored as a sequence of records—one record for each indentifier, constant, or variable which is referenced but not defined in the source, as well as records for items defined to be accessible from other procedures.

The following Pascal-like declaration describes one record within Linker information:

```
LITYPES = (EOFMARK, UNITREF, GLOBREF, PUBLREF,
PRIVREF, CONSTREF, GLOBDEF, PUBLDEF, CONSTDEF,
EXTPROC, EXTFUNC, SEPPROC, SEPFUNC, SEPPREF,
SEPFREF); {Linker information types}

OPFORMAT = (WORD,BYTE,BIG); {label size}

LCRANGE: 1..MAXLC; {currently MAXINT (32767)}

PROCRANGE: 1..MAXPROC; {currently 160}

LIENTRY = RECORD

  NAME: PACKED ARRAY[0..7] OF CHAR;
        {name of unit, procedure, or variable
        symbol}

  CASE LITYPE: LITYPES OF

    GLOBREF,  {reference to a global address}
    PUBLREF,  {reference to a host program
              variable}
    PRIVREF,  {reference to private variables in a
              host activation record}
    CONSTREF, {reference to a global constant}
    UNITREF,  {reference to a regular unit}
    SEPPREF,  {unused}
    SEPFREF:  {unused}
      (FORMAT:  OPFORMAT;
       NREFS:  INTEGER;
       NWORDS: LCRANGE;
       POINTERLIST: ARRAY [1..((NREFS-1) DIV 8)+1]
        OF ARRAY [0..7] OF INTEGER);
          {segment-relative pointers}

     GLOBDEF:  {global address definition}
       (HOMEPROC: PROCRANGE;
        ICOFFSET: LCRANGE);

     PUBLDEF:  {host program variable definition}
       (BASEOFFSET: LCRANGE);

     CONSTDEF: {host program constant definition}
       (CONSTVAL: INTEGER);

     EXTPROC,  {EXTERNAL procedure declaration}
     EXTFUNC,  {EXTERNAL function declaration}
     SEPPROC,  {separate assembly procedure}
     SEPFUNC:  {separate assembly function}
       (SRCPROC: PROCRANGE;
        NPARAMS: INTEGER);

     EOFMARK:  {end-of-file mark}
       (NEXTBASELC: LCRANGE;
        PRIVDATASEG: SEGNUMBER);

  END;
END;
```

Linker Information Fields

The **Linker information types** GLOBREF, PUBLREF, PRIVREF, CONSTREF, and UNITREF, all have similar fields. The FORMAT field may be BIG, BYTE or WORD, and specifies the format of the P-machine **operand** that refers to the entity given by the NAME array (see Chapter 5, Instruction Formats for a description of these formats). The NREFS field specifies the number of references to this entity in the code segment; there will be an equivalent number of entries in the POINTERLIST array. The NWORDS field specifies the amount of space, in words, to be allocated for PRIVREF Linker information types; the NWORDS field is ignored for all other Linker information types.

The **POINTERLIST** array is a list of pointers into the code segment, each of which points to a location within the code segment where there is a reference to the entity specified by the NAME array. The locations are given as absolute byte addresses within the code segment. The POINTERLIST array is composed of records of eight words, but only the first ((NREFS−1) MOD 8)+1 words of the last record are used. All unused words in each array are zeroed.

Global Address Linker Information Types

Separate assembly-language procedures and functions can share data structures and subroutines by means of the .DEF, .REF, .PROC, and .FUNC Assembler directives. These directives cause the Assembler to generate information that the Linker uses to resolve external references between **separate procedures** and **functions** in the same assembly or between procedures and functions assembled separately. Each entity referenced by a .REF Assembler directive results in a GLOBREF Linker information type entry that designates fields to be updated by the Linker. Each entity defined by a .DEF, .PROC, or .FUNC Assembler directive results in a GLOBDEF Linker information type entry that provides the Linker with the values to fix the .REF references.

The GLOBREF Linker information type is used to link addresses between assembled procedures. The FORMAT field is always WORD. The NREFS field specifies the number of pointers in the POINTERLIST array (each of which points to a different reference).

The GLOBDEF Linker information type defines the location of an entity in an assembled procedure. The HOMEPROC field contains the number of the procedure that defines the entity specified by the NAME array. The ICOFFSET field specifies the location within the named procedure where the entity is defined. The location is given as a byte offset, relative to the start of the procedure. There is no POINTERLIST array associated with a GLOBDEF Linker information type.

As a program is linked, the Linker picks up each address defined explicitly by .DEF and implicitly by .PROC and .FUNC, and fixes up each reference to it in other procedures. The Linker must insert the final segment offset of the address in all words pointed to by the POINTERLIST array.

Host-Communication Linker Information Types

The Assembler directives .CONST, .PUBLIC, and .PRIVATE enable an assembly-language procedure or function to share addresses and data space with the host program that calls it. Data values and locations are referred to by name in both the host program and the called procedure or function. Each entity referenced by a .CONST, .PUBLIC, or .PRIVATE Assembler directive results in a CONSTREF, PUBLREF, or PRIVREF Linker information type entry, respectively, that designates fields to be fixed up by the Linker. Each entity defined by a CONSTANT or VARIABLE declaration results in a CONSTDEF or PUBLDEF Linker information type entry, respectively, that provides the Linker with the values to fix references. As a program is linked, the Linker picks up each entity defined by .CONST, .PUBLIC, and .PRIVATE, and fixes up each reference to it in other procedures. The Linker must insert the final segment offset of the address in all words pointed to by the POINTERLIST array.

The PUBLREF Linker information type is used to link global variables in the activation record of a host program to assembly-language procedures or regular units (activation records are explained in the next chapter). The PUBLREF Linker information type results from a .PUBLIC directive in an assembly-language procedure or from use of variables declared in the INTERFACE of regular units. The NAME array specifies a variable that is referenced in the segment, and defined as a global variable in the host program. The FORMAT field is WORD for assembly-language procedures, and BIG for regular units. The NREFS field specifies the number of pointers in the POINTERLIST array (each of which points to a different reference). The Linker must add the offset of the referenced identifier to all words pointed to by the POINTERLIST array.

The PUBLDEF Linker information type declares a global variable in the host program. A PUBLDEF Linker information type is generated for each global variable in the host program that appears in a VAR declaration. The BASEOFFSET field specifies the location of the variable specified by the NAME array within the activation record of the host program that contains it. The location is given as a word offset, relative to the start of the **data area**. There is no POINTERLIST array associated with a PUBLDEF Linker information type.

The CONSTREF Linker information type is used to link constants in an assembled procedure to a global constant in the host program. The CONSTREF Linker information type results from a .CONST directive in an assembly-language procedure. The NAME array specifies a constant that is referenced in the segment, and defined as a global constant in the host program. The FORMAT field is WORD. The NREFS field specifies the number of pointers in the POINTERLIST array (each of which points to a different reference). The Linker must place the constant value into all locations pointed to by the POINTERLIST array.

The CONSTDEF Linker information type declares a global constant in the host program. A CONSTDEF Linker information type is generated for each global constant in the host program that appears in a CONSTANT declaration. The CONSTVAL field contains the value of the declared constant. There is no POINTERLIST array associated with a CONSTDEF Linker information type.

The PRIVREF Linker information type is used to indicate a reference to variables of an assembly-language procedure or regular unit, to be stored in the host program's global data area, and yet be inaccessible to the host program. The PRIVREF Linker information type results either from a .PRIVATE directive in assembly language, or by the use of global variables declared in the IMPLEMENTATION of regular units. The FORMAT field is always WORD. The NWORDS field specifies the amount of space, in words, to be allocated. The NREFS field specifies the number of pointers in the POINTERLIST array. The Linker must add the offset of the start of the allocated area within the global data area to all words pointed to by the POINTERLIST array.

The UNITREF Linker information type is used to link references between regular units. The NAME array specifies the name of a regular unit that is

referenced within another regular unit. The FORMAT field is always BYTE. The NREFS field specifies the number of pointers in the POINTERLIST array (each of which points to a different reference). The Linker must insert the final segment number of the references unit in all locations pointed to by entries in the POINTERLIST array.

Procedure and Function Linker Information Types

Separate assembly-language procedures and functions are referenced via EXTERNAL declarations in the calling segment. The Linker information types EXTPROC, EXTFUNC, SEPPROC, and SEPFUNC are used to link procedures and functions between segments. Each .PROC or .FUNC entity referenced by a PROCEDURE...EXTERNAL declaration results in an EXTPROC or EXTFUNC Linker information type entry, respectively, that designates fields to be fixed up by the Linker. All procedure or function code that begins with .PROC or .FUNC results in a SEPPROC or SEPFUNC Linker information type entry, respectively, that provides the Linker with the values to fix references. As each procedure or function is linked, the Linker picks up each procedure number and parameter size declared in the separate procedure or function, and transfers it to each external reference of that same procedure or function.

The SRCPROC field specifies the procedure number of the referenced or declared procedure. The NPARAMS field specifies the number of words of parameters indicated in the .PROC or .FUNC directive. There is no POINTERLIST array associated with EXTPROC, EXTFUNC, SEPPROC, or SEPFUNC Linker information types.

Miscellaneous Linker Information Types

The EOFMARK Linker information type indicates the end of Linker information records. Additionally, if the segment is of the host program, the NEXTBASELC field indicates the number of words in the host program's global data area. If the segment is an intrinsic unit code segment, the PRIVDATASEG field contains the segment number of the associated data segment.

3

The P-Machine

3
The P-Machine

The previous chapter discussed the static structure of program codefiles on disk and in memory. This chapter discusses the dynamic structure of program code as it is being executed in memory.

System Memory Use

Figures 3-1 and 3-2 are diagrams of the Apple III's memory when running under the Apple Pascal system. These memory maps are specific to the Apple III, and do not apply to any other computer. They are provided for your information only: a primary task of the Apple Pascal system is to eliminate the necessity for the programmer to know anything about specific memory addresses and use.

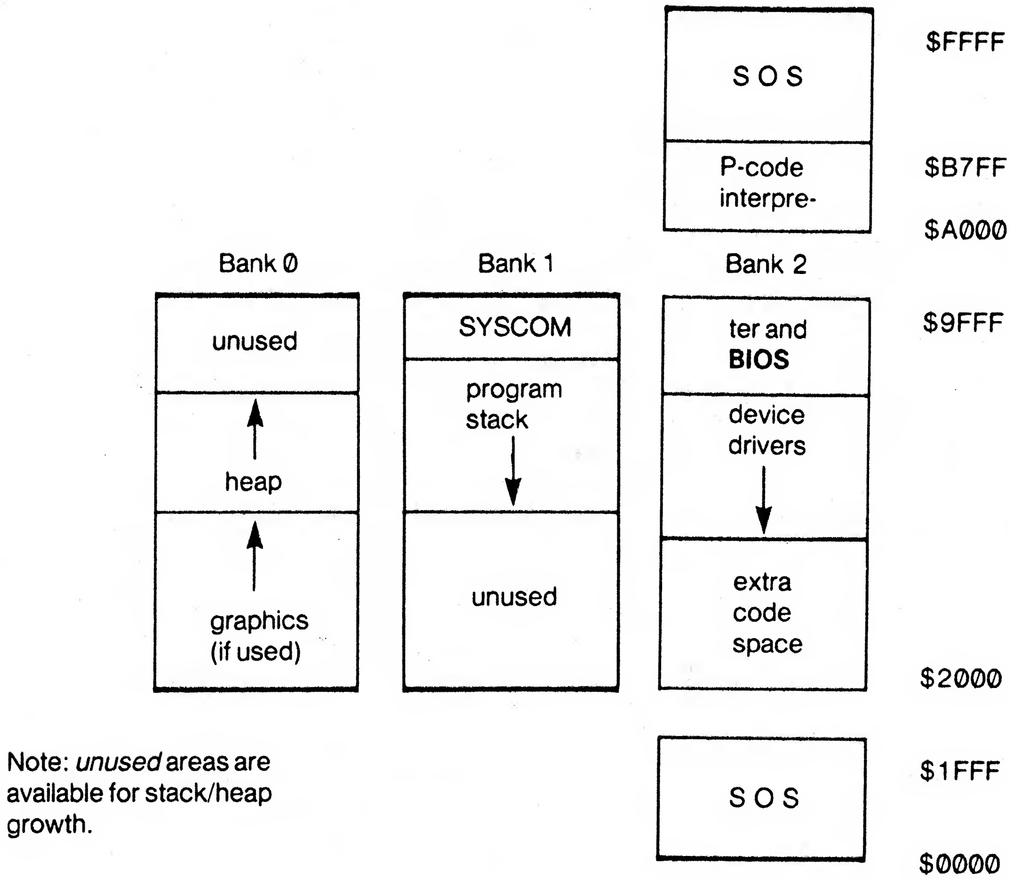
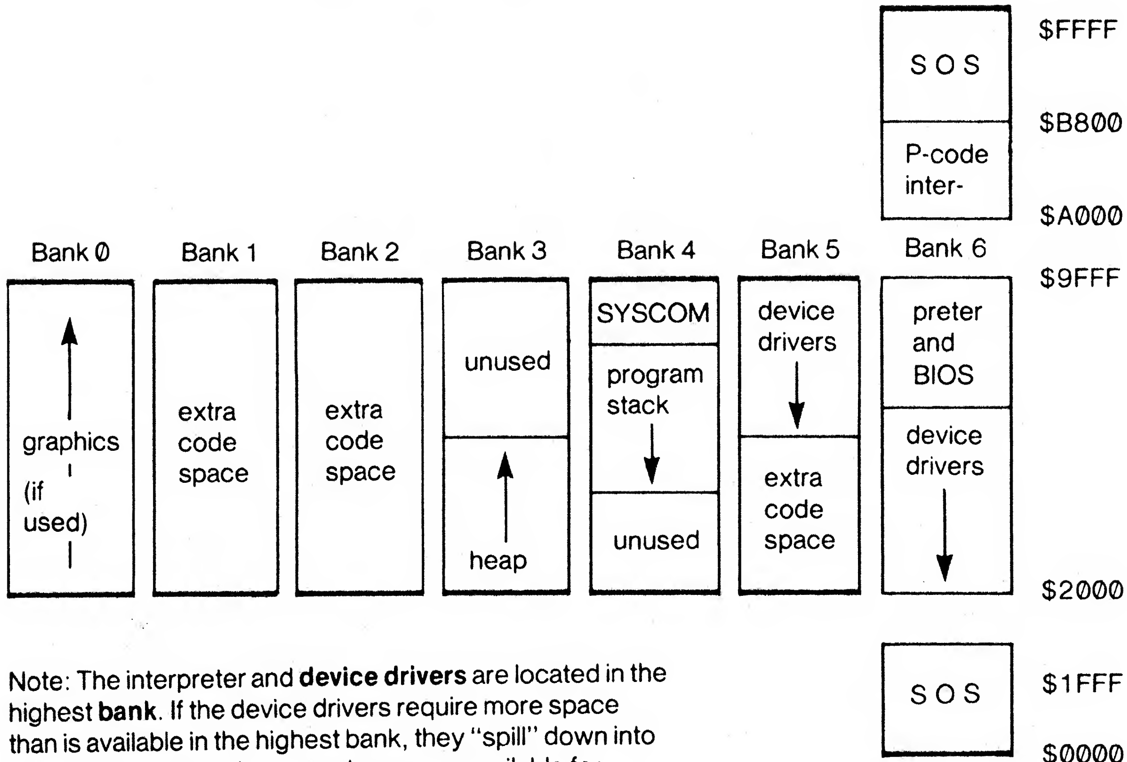


Figure 3-1. Typical Memory Map
of a 128K Apple III Using Apple Pascal



Note: The interpreter and **device drivers** are located in the highest **bank**. If the device drivers require more space than is available in the highest bank, they "spill" down into the next highest bank. *unused* areas are available for stack/heap growth.

Figure 3-2. Typical Memory Map of a 256K Apple III Using Apple Pascal

The P-Machine

The Apple III Pascal **pseudo-machine** or **P-machine**, a version of the UCSD Pascal P-machine, is the software-generated **device** that executes P-code as its machine language. Every computer operating under a form of UCSD Pascal has been programmed to "look like" this common P-machine, or a related variant, from the viewpoint of a program being executed. The P-machine has an **evaluation stack**, several registers, and a **user memory**. The user memory contains the **program stack**, the **heap**, and **extra code space** where program code can be stored (Figure 3-3). Each of these structures is discussed in detail below.

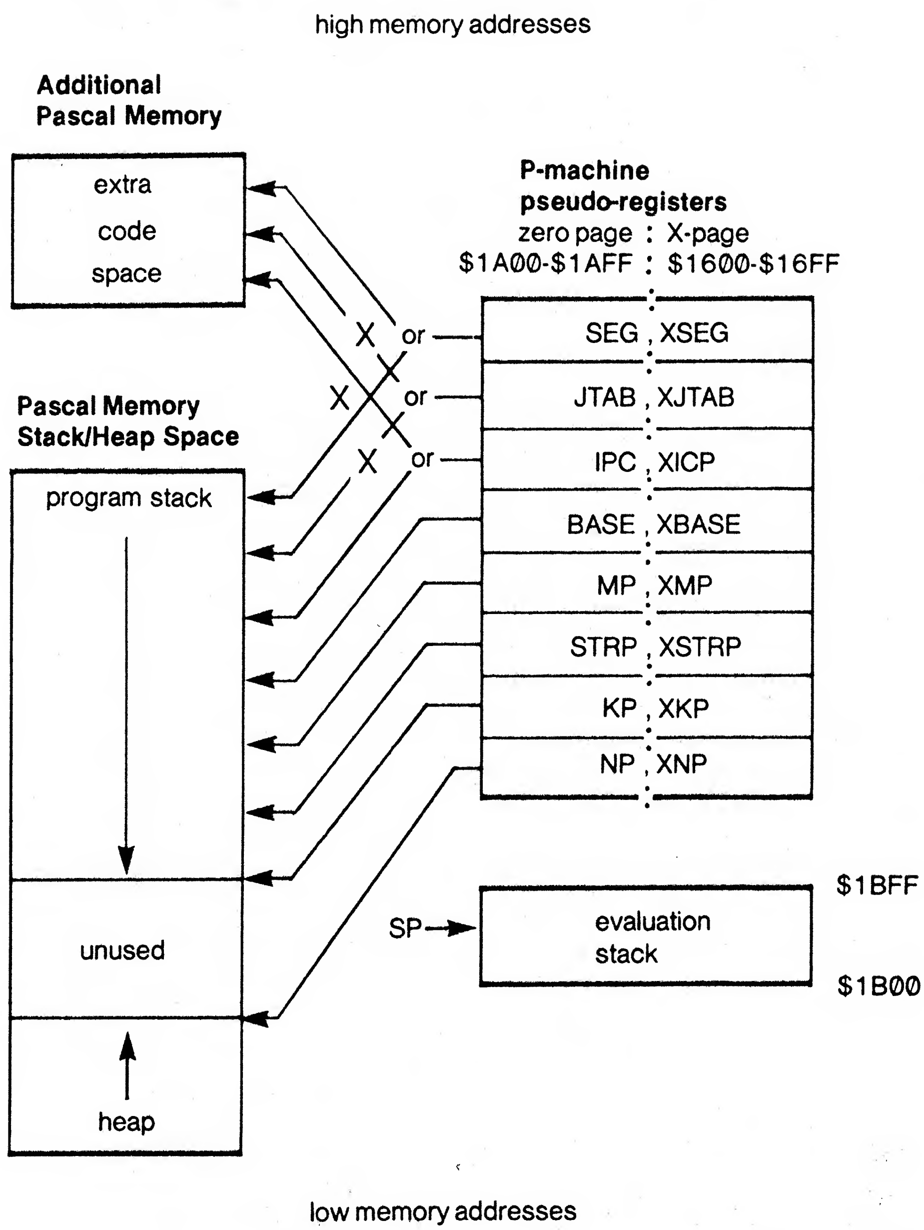


Figure 3-3. The P-Machine Model

Note: In Figure 3-3 pointers to P-code are shown pointing both up and down, because P-code may be in either the stack or extra code space. Pointers to data are shown pointing down only, because data is stored only in the **stack/heap space**.

The P-machine supports:

- Variable addressing, including strings, byte arrays, packed fields, and **dynamic variables**
- Logical, integer, real, set, array, and string, top-of-stack arithmetic and comparisons
- Multi-element structure comparisons
- Branches
- Procedure and function calls and returns, including overlayable procedures
- Miscellaneous procedures used by system and user programs

The P-machine uses 16-bit words, with two 8-bit bytes per word. Words consist of two bytes, of which the lower, even-address byte is least significant (Figure 3-4). The least significant bit of a word is bit 0, the most significant is bit 15.

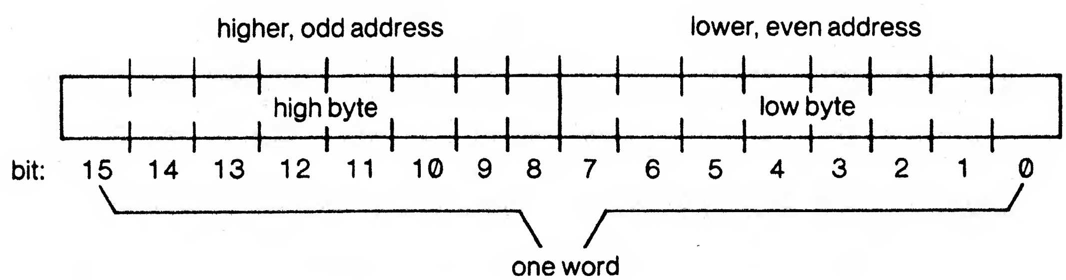


Figure 3-4. Relationship of Words and Bytes

The Evaluation Stack

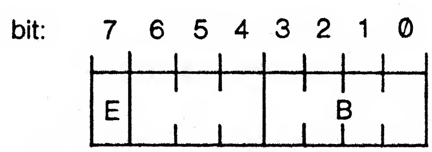
In the Apple III, the evaluation stack uses a portion of the relocated 6502 hardware stack, starting at memory location $1BFF and growing downward to location $1B00. It is used for passing parameters, returning function values, and as an operand source for many P-machine instructions. When an instruction is said to *push* an item, that item is placed on the top of the evaluation stack (the evaluation stack grows downward). The evaluation stack is extended by loads and is reduced by stores and most arithmetic operations.

Enhanced Indirect Addressing

Enhanced indirect addressing is the method used in the Apple III to extend its memory addressing beyond 64K bytes. It involves the use of 6502 indirect-X and indirect-Y addressing modes and depends on hardware interaction between the **zero page** and its corresponding extension page (**X-page**).

SOS permanently assigns locations $1A00 through $1AFF as the user zero page, and the hardware automatically associates locations $1600 through $16FF as the X-page. Although the zero-page data actually resides at locations $1A00 through $1AFF, instructions that refer to the zero page still use address values $00 through $FF. Most Apple III instructions behave exactly like their 6502 counterparts, except for indirect-X and indirect-Y instructions. Depending on the values in the X-page, these instructions can invoke enhanced indirect addressing.

Consider an indirect-X or indirect-Y reference through zero-page location n. As usual for the 6502, zero-page locations n and n+1 are expected to contain the operand address (disregarding indexing by X or Y, for the moment). Since the zero page is mapped, this operand address is actually stored at locations $1A00+n and $1A01+n, with the least-significant byte at the lower address. Location $1601+n contains the **X-byte** for this addressing operation. The X-byte is interpreted as follows:



Bit 7 is the enhanced-addressing bit, or E-bit. If it is zero, normal 6502 addressing occurs and the rest of the X-byte is ignored. Normal 6502 addressing means addressing in the 64K address space consisting of a lower 8K portion followed by the currently switched-in 32K bank followed by an upper 24K portion.

The normal user should not access the lower 8K or upper 24K, because these are occupied by SOS.

If bit 7 of the X-byte is one, enhanced indirect addressing occurs. The four-bit field B specifies a bank pair consisting of banks B and B+1. These two banks together make up a continuous 64K address space. The address word stored in zero page is taken as the address of a location in this 64K bank pair, regardless of which bank is currently switched in.

Locations $0000 to $00FF (the zeroth page) in each bank pair are actually mapped into the current user zero page (locations $1A00 to $1AFF). These locations should be addressed using ordinary zero-page addressing.

Registers

The Apple III P-machine uses eight **pseudo-registers**, and the hardware stack pointer (Figure 3-3). All registers are pointers to **word-aligned** structures, except the IPC register, which is a pointer to **byte-aligned** structures.

Because the Apple III uses an enhanced-indirect addressing architecture, each psuedo-register (except the SP register), consists of two parts. One part is a 16-bit pointer on zero page, and the other is a corresponding X-byte on X-page. Thus each register (except SP) consists of two components; for example, IPC and XIPC. The psuedo-registers are:

SP: **evaluation Stack Pointer**. This register contains a pointer to the current *top* of the evaluation stack (one byte below the last byte in use). It is actually the Apple III hardware stack pointer.

IPC, XIPC: **Interpreter Program Counter**. This register contains the address of the next instruction to be executed in the currently-executing procedure.

SEG, XSEG: **SEGment pointer**. This register points to the highest word of the procedure dictionary of the segment to which the currently-executing procedure belongs.

JTAB, XJTAB: **Jump TABle pointer**. This register contains a point to the highest word of the attribute table in the procedure code of the currently-executing procedure. (Attribute tables are explained in Chapter 2.)

MP, XMP: **Markstack Pointer**. This register contains a pointer to the MSSTAT field, in the **markstack** of the currently executing procedure. Local variables in the activation record of the current procedure are accessed by indexing off of the location pointed to by the MP register. (Markstacks are explained later in this chapter).

BASE, XBASE: **BASE procedure pointer**. This register contains a pointer to the MSSTAT field of the activation record of the most recently invoked base procedure (lexical level 0 or 1). Global (lex level 0) variables are accessed by indexing off of the location pointed to by the BASE register. (Activation records are explained later in this chapter.)

STRP, XSTRP: **STRing Pointer**. This register is a pointer to the first element of the linked list of packed arrays of characters and strings on the stack. Whenever the P-machine executes an LPA or LSA instruction (see Chapter 5), and the literal packed array or string constant contained in the instruction is not already on the program stack, the P-machine pushes it onto the program stack and links it into the list pointed to by this pseudo-register.

KP, XKP: **program stacK Pointer**. This register contains a pointer to the lowest byte of the lowest word actually in use on the program stack. The program stack starts in high addresses of user memory and grows downward toward the heap.

NP, XNP: **New Pointer**. This register contains a pointer to the current top of the heap (one byte above the last byte in use). The heap starts in low addresses of user memory and grows upward toward the program stack. It contains all dynamic variables. It is extended by the standard Pascal procedure `'new'`, and is cut back by the standard procedure `'release'`.

Extra Code Space

The segments of an executing program may be loaded by the interpreter into several different areas of memory. Segments are preferentially loaded in areas labeled *extra code space* in Figures 3-1 through 3-3, so that the space in the stack/heap area is not needlessly consumed. Only when the *extra code space* areas are filled, are segments loaded onto the stack. Segments are never loaded into the *unused* area between the stack and heap.

The Program Stack and the Data Heap

The operating system uses two dynamic structures called the program stack and the heap to store memory-resident data of an executing program. The program stack and heap reside in the same bank pair. The program stack is used to store **automatic variables**, strings, packed arrays, bookkeeping information about procedure and function calls, and code segments if there is no available extra code space. The heap is used to store dynamic variables.

Figure 3-5 is a diagram of the Pascal program stack and heap with four active procedures.

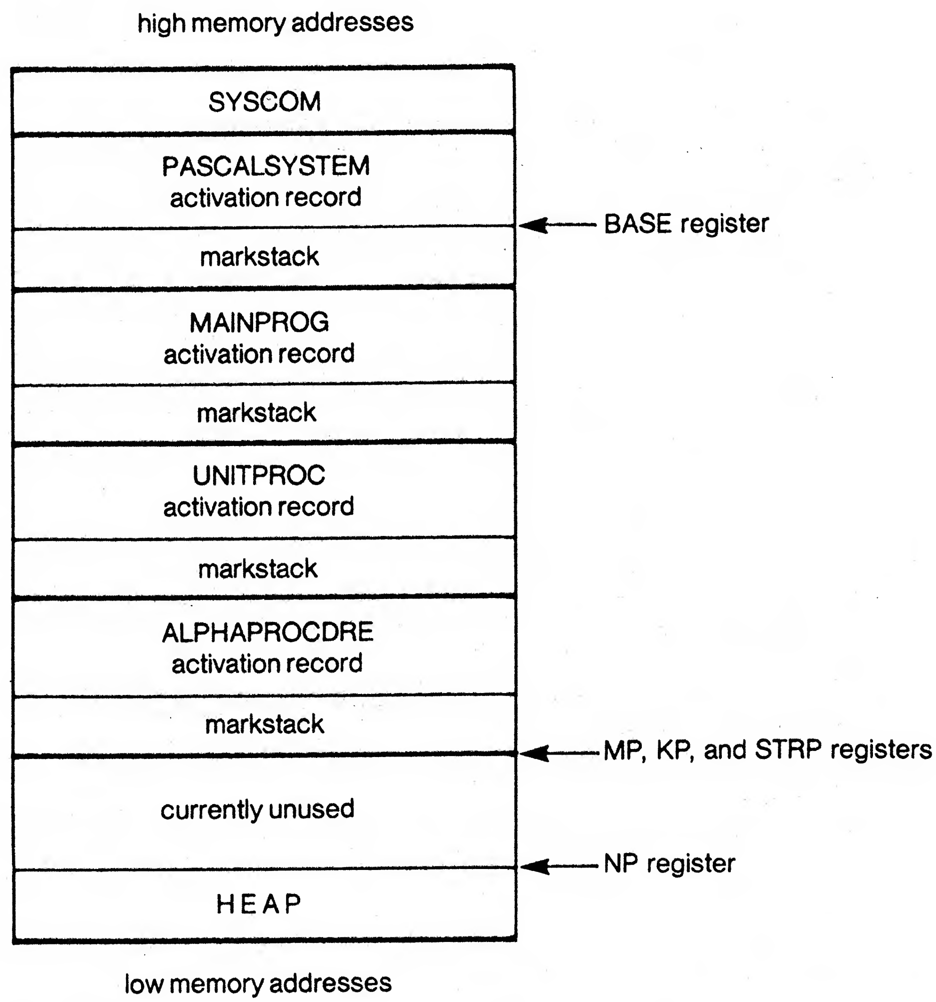


Figure 3-5. The Program Stack and Heap With Four Active Procedures

SYSCOM

The operating system and the P-machine exchange information via the system communications area (also called **SYSCOM**) at the bottom (high addresses) of the stack. SYSCOM is accessible to both assembly-language procedures in the interpreter and (as if it were part of the Pascal system global data) to system procedures coded in Pascal. SYSCOM serves as an important communication link between these two levels of the system. The fields in SYSCOM relevant to communication between the operating system and the P-machine are:

IORSLT: This field contains the error code returned by the last activated or terminated I/O operation (see Volume 2 of the *Apple III Pascal Programmer's Manual* for a list of I/O Error messages).

XEQERR: This field contains the error code of the last execution error (see Volume 2 of the *Apple III Pascal Programmer's Manual* for a list of execution error messages).

BOMBP: This field contains a pointer to the activation record of the procedure that caused the execution error.

BOMBSEG, BOMBPROC, BOMBIPC: These fields contain the segment number, procedure number, and IPC value when an execution error occurs.

SYSUNIT: This field contains the Pascal volume number of the device from which the operating system was booted (usually the boot disk drive, volume 4).

GDIRP: This field contains a pointer to the most recent Apple II format disk directory read in, unless dynamic allocation or deallocation has taken place since then (see the MRK, RLS, and NEW instructions in Chapter 5). Disk directories are read into a temporary buffer directly above the heap. (Not used for SOS-format directories.)

Segment Table: The segment table is a record that contains information needed by the P-machine to read code segments into memory or to allocate space for data segments.

THE SEGMENT TABLE

Every code segment has a name, but when a given segment references another during the execution of a program, it refers not to the segment's name, but to the segment's number. The interpreter uses the segment number as an index into the segment table, which contains an entry for each segment in the program (Figure 3-6). The segment table entries are indexed by segment number; each entry contains information needed to load the segment from the codefile on disk into memory. The segment table is a dynamic structure of SYSCOM, but is somewhat analagous to a segment dictionary, in that it is used to locate segments on disk.

The segment table is located in the higher addresses of the SYSCOM area, at the bottom of the program stack. It contains entries for:

- the segments of the Pascal operating system itself (numbers 0, 2...6, 59...63)
- each segment in the segment dictionary of the host program codefile
- each intrinsic unit code and data segment in library files linked with the host program

No two segments in an executing program can have the same number since the numbers are used to index the segment table. The segment table has space for up to 64 entries. Since the system can use 11, this means that 53 entries are left for the program to use.

Remember that a program codefile contains 16 or fewer segments; any excess over 16 must be in either a program library file, a SYSTEM.LIBRARY file, or library files specified in a library name file.

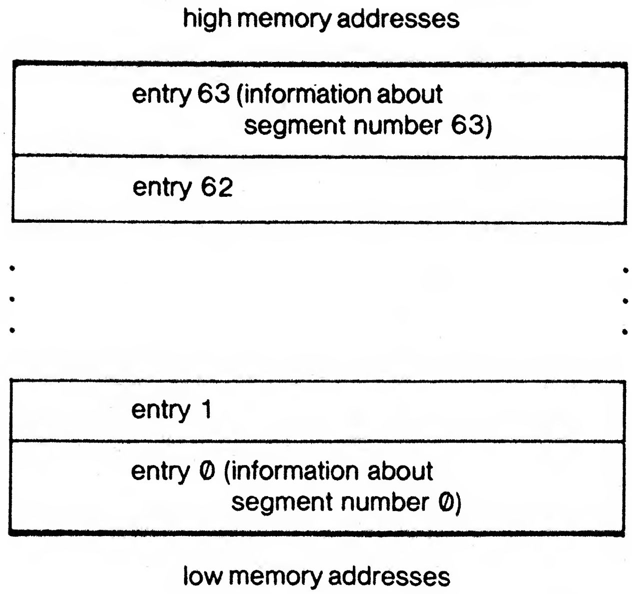


Figure 3-6. The Segment Table

Activation Records

When a procedure is called, the code segment containing that procedure code is loaded by the interpreter if it is not already present in memory. An activation record for the procedure is built on top of the program stack each time the procedure is called (Figure 3-7). Only code segments require activation records, data segments do not. The activation record for a procedure consists of:

- the markstack, which contains addressing context information (**static links**), and information on the calling procedure's environment
- space for storing the value returned by the procedure, if the procedure is a function
- space for parameters passed to the procedure when it is called
- space for the local variables of the procedure

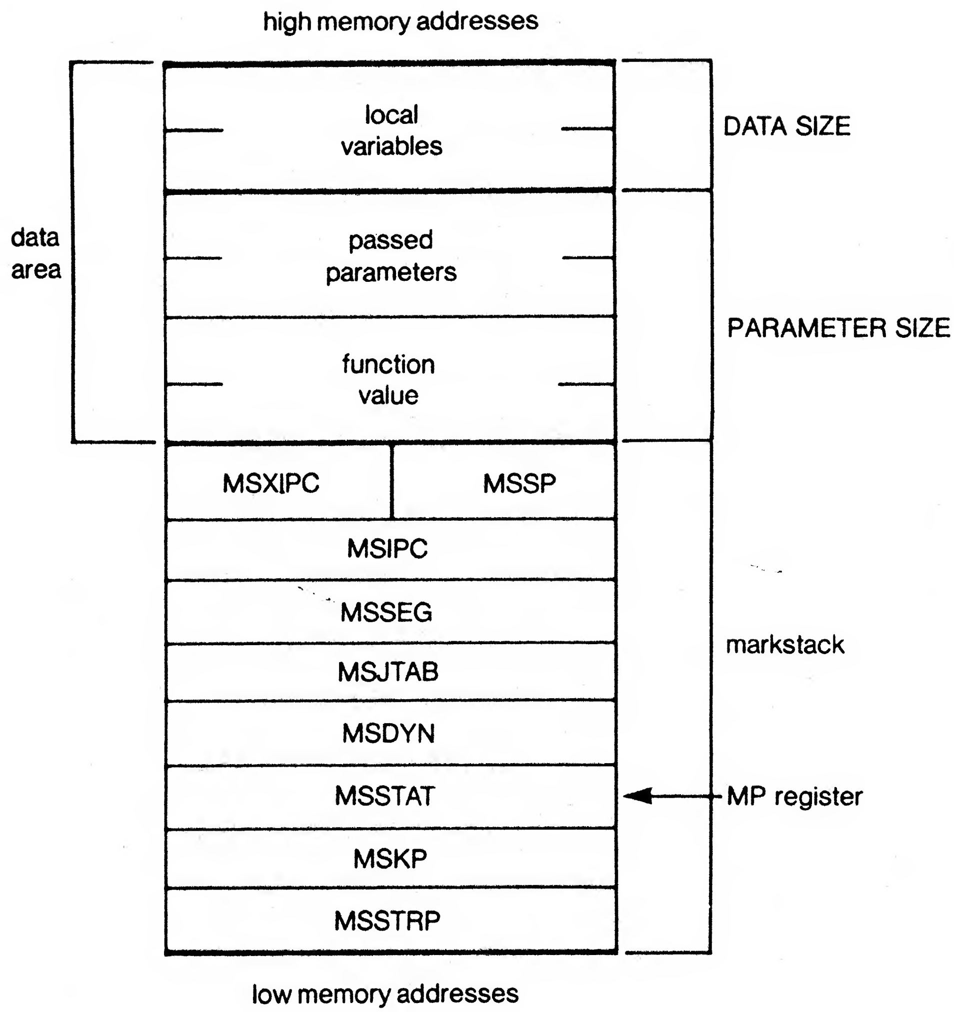


Figure 3-7. An Activation Record

Space is allocated in the higher addresses of the activation record for variables local to the procedure. The variable space is allocated in the reverse order that the variables are declared (exception: variables in a variable list are allocated space in forward order). For example, the statements

```
VAR I,J: INTEGER;
    BOOL: BOOLEAN;
```

will cause space in the activation record to be allocated as shown in Figure 3-8.

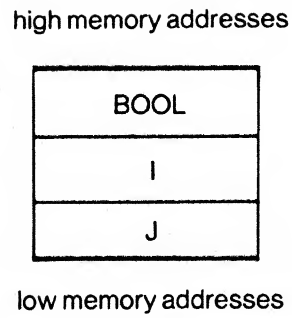


Figure 3-8. The Order of Local Variable Allocation in an Activation Record

Space for parameter passing is allocated below the local variable space. If the procedure is a function, space is also reserved (below the parameter space) for storing the value returned by the function. The order of passed parameters is discussed in Chapter 4. A description of the format of variables in activation records is given in Chapter 5.

Local variables in the activation record of an active procedure are accessed by indexing off of the location pointed to by the MP, XMP register. Global variables in the activation record of an active procedure are accessed by indexing off of the location pointed to by the BASE, XBASE register.

When a procedure is terminated, its activation record is removed from the stack.

Markstacks

The lower portion of the activation record is called a markstack. When a procedure call is made, the current values of the system psuedo-registers that characterize the operating environment of the calling procedure are stored in the markstack of the called procedure. This allows the system registers to be restored to precall conditions when control is returned to the calling procedure.

A procedure call causes the operating environment that existed in the system registers just at the time of the procedure call to be stored in the fields of the called procedure's markstack in the following manner:

System registers		Markstack fields	
SP	→	MSSP	(MarkStack evaluation Stack Pointer)
IPC	→	MSIPC	(MarkStack Interpreter Program Counter)
XIPC	→	MSXIPC	(MarkStack X-byte of Interpreter Program Counter)
SEG	→	MSSEG	(MarkStack SEGment pointer)
JTAB	→	MSJTAB	(MarkStack Jump TABle pointer)
MP	→	MSDYN	(MarkStack DYNamic link)
KP	→	MSKP	(MarkStack program stacK pointer)
STRP	→	MSSTRP	(MarkStack STRing Pointer)

The MSDYN field of a markstack contains a pointer to the MSSTAT field in the markstack of the procedure that called the new procedure. The combined MSDYN fields of all markstacks form a **dynamic chain** of links that describe the "route" by which the new procedure was called.

The MSSTAT field of a markstack contains a pointer to the MSSTAT field in the most recent markstack of the procedure that is the lexical parent of the called procedure. The interpreter "knows" which procedure is the lexical parent, by looking up the **static chain** until it encounters a procedure whose lexical level is one less than the lexical level of the current procedure. The combined MSSTAT fields of a group of markstacks form a static chain of links that describe the lexical nesting of the called procedure.

The values of the XSEG and XJTAB registers are not stored on the markstack because they are equivalent to XIPC. The XKP, XMP, and NP, XNP registers are not stored because they do not change during a procedure call. The BASE, XBASE registers are not stored on the markstack because their values are related only to base procedure calls.

After building the new procedure's activation record on the program stack, new values for the IPC, XIPC, SEG, XSEG, JTAB, XJTAB, KP, STRP, XSTRP, and MP registers, are established. The registers are updated as follows:

- The SP register is unchanged, and remains pointing to the top of the evaluation stack.
- The KP, XKP register points to the new top of the program stack, just beyond the newly-created activation record.
- The IPC, XIPC register points to the first instruction of the called procedure.
- The SEG, XSEG register points to the procedure dictionary of the code segment that contains the called procedure.
- The JTAB, XJTAB register points to the attribute table of the called procedure.
- The MP, XMP register points to the markstack of the called procedure.
- The STRP, XSTRP register is initialized to NIL (zero).
- If the called procedure has a lexical level of -1 or Ø, the contents of the BASE register are saved on the evaluation stack, and the BASE register is set to the value of the MP register.

Each time a procedure is called, another activation record is added to the program stack. Once again the register values and the appropriate **static link** and **dynamic link** are stored in the new markstack, and the system registers are then updated. Note that the SEG register always points to the procedure dictionary of the segment that contains the procedure (and not the segment that called the procedure).

Once the code for a procedure has been loaded into memory, each further invocation of the same procedure causes only an activation record to be added to the program stack (the code is not loaded again).

When a return from a procedure occurs, the information in the markstack fields is transferred to the system registers, and the activation record of the inactive procedure is removed from the stack.

Additional information on procedure calls, and the relation of attribute tables to activation records, can be found in the section Procedure and Function Calls in Chapter 5.

4

Assembly-Language Programming

4 Assembly-Language Programming

Calling Assembly Procedures and Functions

A separate procedure or function is written in assembly language as a .PROC or .FUNC. The assembled code is assumed to be in a codefile which will be linked into the host program before execution. Within the host program, the EXTERNAL procedure or function must be declared by a standard PROCEDURE or FUNCTION heading followed by the keyword EXTERNAL. For example,

```
PROCEDURE MAKESCREEN (INDEX: INTEGER); EXTERNAL;
```

declares the procedure MAKESCREEN as an EXTERNAL assembly-language procedure, with one parameter of type integer.

Calls to EXTERNAL procedures use standard Pascal syntax, and the Compiler checks that each call agrees in type and number of parameters with the declaration for that procedure. It is the programmer's responsibility to ensure that the assembly-language procedure is compatible with the EXTERNAL declaration of the host program. The Linker checks only that the number of words of parameters in the host program's EXTERNAL declaration and in the separate procedure's .PROC or .FUNC declaration are the same.

Variable parameters in EXTERNAL procedures and functions can be declared without any type.

Passing Parameters to Assembly Procedures

When the host program executes a call to an EXTERNAL procedure or function, the parameters to be passed are pushed onto the evaluation stack in the order they are encountered in the host program's calling statement: the first parameter is pushed onto the stack (high byte first), then the second parameter, and so on. When all the parameters have been passed, the host program's return address is pushed onto the stack (high byte first) (Figure 4-1). In addition, if the procedure is a function, the host program pushes two words (four bytes) of zeros onto the evaluation stack after any parameters are pushed and before the return address is pushed.

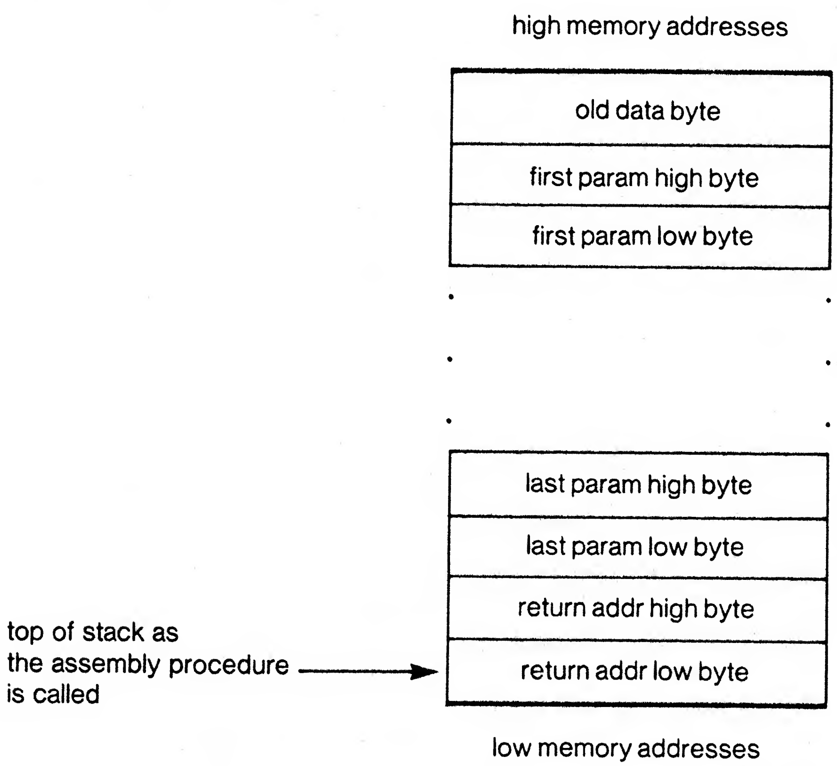


Figure 4-1. Order of Parameters on the Stack

The passed parameters are available on the stack in reverse order: the last one passed is at the top of the stack. For example, the function call

```
FUNCTION MULT3(I,J,K:INTEGER); EXTERNAL;
```

causes I to be pushed onto the stack first, then J, then K, then four unused bytes, and finally the return address (Figure 4-2). Then the function is called.

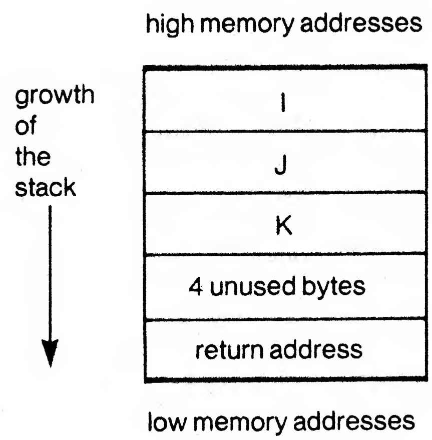


Figure 4-2. The Order of Parameters on the Stack Just Prior to Execution of a Function

Long integers and sets are passed as the number of words used in the host program. Again, each word is pushed onto the stack high byte first. After a long integer or set, a word indicating the number of words passed is pushed onto the stack. Strings, records, arrays, and VAR parameters are passed by address, high byte first. Recall that the host program's EXTERNAL declaration may declare a VAR parameter without a type, which allows a parameter of indeterminate size to be passed by address. Below are listed the various ways that different parameter types are represented on the stack.

Passing Mode	Parameter Type	Representation on Stack
Pass by reference (VAR)	all types	A 2-byte pointer to the value, with the high byte pushed first
Pass by value	integer, subrange, enumerated type	A 2-byte value, with the high byte pushed first
	real	A 4-byte value, with the high byte pushed first
	set	1 to 32 words (representing the set value) are pushed first. Then a word specifying the number of words in the set is pushed (high byte first).
	long integer	1 to 9 words (representing the long integer value) are pushed first. Then a word specifying the number of words in the long integer value is pushed (high byte first).
	string, record, array	A 2-byte pointer to the value, with the high byte pushed first

Examples of Assembly-Language Procedures

The TIMES2 function in the assembly-language example below uses parameter-passing by value.

```
                            ;sample assembly language for
                            ;FUNCTION TIMES2(DATA:INTEGER):
                            ; INTEGER;

        .FUNC   TIMES2,1    ;one word of parameters
RETURN  .EQU    Ø           ;temp store return address

        PLA                 ;save host segment return address
        STA     RETURN
        PLA
        STA     RETURN+1
        PLA                 ;discard four unused bytes
        PLA                 ;(only necessary for functions)
        PLA
        PLA

        PLA                 ;least significant byte (lsb) of data
        ASL     A           ;times 2
        TAX                 ;save in X
        PLA                 ;most significant byte (msb) of data
        ROL     A           ;times 2, with carry
        PHA                 ;move msb to evaluation stack

        TXA                 ;restore lsb to accumulator
        PHA                 ;move lsb to evaluation stack

        LDA     RETURN+1    ;restore host segment return address
        PHA
        LDA     RETURN
        PHA
        RTS                 ;return to calling segment
```

The function first removes the return address from the stack and saves it in the location RETURN. After discarding the four unused bytes added to the stack because the host program was calling a function, the function then picks up the data word, one byte at a time. When it is finished, the function pushes the result back onto the stack, followed by the return address.

The SETZERO procedure in the assembly-language example below uses parameter passing by reference.

```
                                ;sample assembly language for
                                ;PROCEDURE SETZERO (VAR I:INTEGER);
                                ; EXTERNAL;

        .PROC   SETZERO,1
RETURN  .EQU    Ø               ;temp store return address
DATADR  .EQU    ØEØ
        PLA                     ;save host segment return address
        STA     RETURN
        PLA
        STA     RETURN+1
        PLA                     ;Put address of parameter
        STA     DATADR          ;I into locations $ØEØ-ØE1
        PLA
        STA     DATADR+1        ;$16E1 is already set to
                                ;Pascal data area x-value

        LDA     #Ø              ;Zero to A
        TAY                     ;...and Y

        STA     @DATADR,Y       ;Store Ø in word pointed
        INY                     ;to by DATADR
        STA     @DATADR,Y

        LDA     RETURN+1        ;restore host segment return
        PHA                     ;address
        LDA     RETURN
        PHA
        RTS                     ;return to host segment
```

Returning From Assembly Procedures

Procedures and functions remove all parameters from the stack before returning. When a procedure terminates, it pushes the return address back onto the stack, and executes an RTS to the calling segment. When a function terminates, it pushes the return value (a scalar, real, or pointer, maximum two words) and the return address back onto the stack, and then executes an RTS to the calling segment.

Temporary and Semipermanent Storage

When you write assembly-language procedures for the Apple III, you must respect the SOS and Pascal conventions concerning register use and calling sequences. All of the 65Ø2 registers are available, and zero-page locations $Ø through $35 are available for storing temporary variables. However, the Apple III Pascal System also uses these locations as temporaries, so you should not expect data to remain there from one procedure execution to the next. You can save variables in nonzero page memory by using the .BYTE or .WORD directives to reserve space in your assembly-language procedure.

Accessing Pascal Data Space

To access stack/heap space, an assembly-language procedure must perform indirect-X or indirect-Y addressing using an appropriate X-byte value. For example, if the stack/heap is in banks 1 and 2, the appropriate X-byte is $81 (the high-order bit set to one enables enhanced indirect addressing and the low-order bits specify the bank pair 1-2).

Since the Pascal subprocedure linkage mechanism only passes two-byte addresses (the X-byte is excluded), it is the programmer's responsibility to make sure the X-byte is properly set. The Pascal system presets locations $16E1, $16E3, $16E5, and so on through $16EF to the X-byte value for Pascal data space at boot time. Thus, assembly-language procedures can copy parameter addresses into locations $EØ-$E1, $E2-$E3, and so on

through $EE-$EF and perform indirect-X or indirect-Y addressing with these zero-page addresses to access the parameters in Pascal data space.

The following example shows how to access .PUBLIC data by using this approach:

```
                                ;sample assembly language for
                                ;PROCEDURE TEST;

        .PROC    TEST
RETURN  .EQU     Ø
DATR    .EQU     ØEØ            ;first pseudo-register
        .PUBLIC  DATA           ;data belongs to the host

        PLA                     ;save host segment return address
        STA      RETURN
        PLA
        STA      RETURN+1

        LDA      ADATA          ;move address into pseudo-
        STA      DATR           ;register
        LDA      ADATA+1
        STA      DATR+1

        LDY      #Ø
        LDA      @DATR,Y        ;load the DATA into the accumulator

        LDY      #1Ø            ;if DATA = PACKED ARRAY[Ø..2Ø] OF
        LDA      @DATR,Y        ;CHAR, this loads DATA[1Ø]

        LDA      RETURN+1       ;restore host segment return address

        PHA
        LDA      RETURN
        PHA
        RTS                     ;return to host segment

ADATA   .WORD    DATA           ;the host's address of DATA
```

Enhanced indirect addressing also occurs in the assembly-language example below. The INCARRAY procedure pulls the return address from the stack and saves it at location RETURN. It then pulls the address of the array from the stack and stores it in the pseudo-register at location $ØØEØ. After getting the remaining parameters from the stack, the procedure uses enhanced indirect addressing (indirect-Y addressing) to modify the array data where it is stored in memory.

```
                                ;sample assembly language for
                                ;PROCEDURE INCARRAY(SIZE:INTEGER;
                                ; VAR DATA: LIST);

        .PROC   INCARRAY,2      ;2 words of parameters
RETURN  .EQU    Ø               ;temp store return address
SIZE    .EQU    2               ;temp store SIZE
PSUEDO  .EQU    ØEØ             ;pseudo-register

        PLA                     ;save host segment return address
        STA     RETURN
        PLA
        STA     RETURN+1
        PLA                     ;lsb of array address
        STA     PSEUDO
        PLA                     ;msb of array address
        STA      PSEUDO+1
        PLA                     ;lsb of SIZE parameter
        STA     SIZE
        PLA                     ;msb of SIZE discard

        LDY     #Ø              ;initialize array index
ALOOP   CLC                     ;clear for add
        LDA     @PSEUDO,Y       ;load byte array
        ADC     #1              ;increment
        STA     @PSEUDO,Y       ;store incremented array byte
        INY                     ;increment array index
        CPY     SIZE            ;test vs. array SIZE
        BCC     ALOOP           ;do while less than

        LDA     RETURN+1        ;restore host segment return address
        PHA
        LDA     RETURN
        PHA
        RTS                     ;return to host segment
```

All parameters that are passed by address must be accessed by enhanced indirect addressing.

5

The P-Machine Instruction Set

5 The P-Machine Instruction Set

Instruction Formats

Instructions for the P-machine consist of one or two bytes, followed by zero to four parameters. Most parameters specify one word of information. There are five basic types of parameters:

UB: Unsigned Byte. Represents a non-negative integer less than 256. The high-order byte of the parameter is implicitly zero.

SB: Signed Byte. Represents an integer from −128 to 127, in two's-complement form. The high-order byte is a sign extension of bit 7 of the low order byte.

DB: **Don't-care Byte**. Represents a non-negative integer less than 128, thus it can be treated as SB or UB.

B: **Big**. This parameter is one byte long when used to represent values in the range 0 through 127, and is two bytes long when used to represent values in the range 128 through 32767. If the value represented is in the range 0 through 127, the high-order byte of the parameter is implicitly zero. If the value represented is in the range 128 through 32767, bit 7 of the first byte is cleared and the first byte is used as the high order byte of the parameter. The second byte is used as the low-order byte.

W: Word. A two-byte parameter, low byte first. Represents values in the range −32768 through 32767.

Any exceptions to these formats are noted in the descriptions of the individual instructions.

Operand Formats

Although an element of a structure in memory may be as small as one bit (as in a packed array of boolean), variables to be operated on by the P-machine are always unpacked into full words. All top-of-stack (**tos**) operations expect their operands to occupy at least one word on the evaluation stack.

Formats of Variables on the Stack

Variables are stored in activation records and on the evaluation stack in the manner described below.

BOOLEAN: One word. Bit Ø indicates the value (Ø=FALSE, 1=TRUE), and this is the only information used by boolean comparisons. However, the boolean operators LAND, LOR, and LNOT operate on all 16 bits, in a bitwise manner.

INTEGER: One word, two's complement notation, capable of representing values in the range −32768..32767.

LONG INTEGER: 3..11 words. A variable declared as INTEGER[n] is allocated ((n+3) DIV 4) + 2 words of memory space. Regardless of the value of a long integer, its actual size remains the same as its allocated size. Each decimal digit of a long integer is stored as four bits of binary-coded decimal. The format of long integers on the stack is as follows:

word Ø (tos):	contains the allocated length, in words.
word 1 (tos−1):	low byte contains the sign (all zeros = positive, all ones = negative); high byte not used.
word 2 (tos−2):	four least significant decimal digits. The low byte contains the two more significant decimal digits (BCD). The high byte contains the two less significant digits.

word n (tos−n): four most significant decimal digits. The low byte contains the two more significant decimal digits (BCD). The high byte contains the two less significant digits.

The format of long integers in activation records is as follows: word 0 is not stored; word 1 is the lowest word in memory; word n is the highest word in memory.

SCALAR (user-defined): One word, in range 0..32767.

CHAR: One word, with the low byte containing a character. The internal character set is *extended* ASCII, with 0..127 representing the standard ASCII set, and 128..255 representing user-defined characters.

REAL: Two words, whose format is defined by the Proposed Standard for Binary Floating-Point Arithmetic, IEEE Task P754, and described in the *Apple III Pascal Programmer's Manual*, Volume 2. In general, the format for 32-bit real numbers is as follows:

Bit	**Item**	**Contained In**
0..15	mantissa	tos
15..22	mantissa	
23..30	exponent	tos−1
31	sign	

POINTER: One or three words, depending on the type of pointer. Pascal pointers (internal word pointers) consist of one word that contains a word address (the address of the low byte of the word). Internal byte pointers consist of one word that contains a byte address. Internal packed field pointers consist of three words:

word 0 (tos):	right bit number of field
word 1 (tos−1):	field width (in bits)
word 2 (tos−2):	word pointer to the word that contains the field

SET: Ø..31 words in an activation record, 1..32 words on the evaluation stack. Sets are implemented as bit vectors, always with a lower index of zero. A set variable declared as SET OF m..n is allocated (n+15) DIV 16 words of memory space. All words allocated in an activation record for a set contain valid information (the set's actual size agrees with its allocated size).

A set on the evaluation stack is represented by a word (tos) specifying the length of the set, followed by that number of words of information. The set may be padded with extra words (to compare it with another set of different size, say), the length word changed to indicate the number of words in the structure when padded. Before being stored back in an activation record, a set must be forced back to the size allocated to it, by issuing an ADJ instruction.

RECORDS and ARRAYS: Any number of words. Arrays are stored in forward order, with higher-indexed array elements appearing in higher-numbered memory locations. Only the address of the record or array is loaded onto the evaluation stack, never the structure itself. Packed arrays must have an integral number of elements in each word, as there is no packing across word boundaries (it is acceptable to have unused bits in each word). The first element in each word has bit Ø as its low-order bit.

STRINGS: 1..128 words. Strings are a flexible version of packed arrays of characters. A STRING[n] declaration occupies (n DIV 2)+1 words of memory space. Byte Ø of a string is the current length of the string, and bytes 1..length(string) contain valid characters.

Format of Constants in P-Code

CONSTANTS: Constant scalars, sets, and strings may be imbedded in the instruction stream, in which case they have special formats.

- All scalars (excluding reals) greater than 127 are represented by two bytes, high byte first.

- All string literals occupy length(literal)+1 bytes of memory space, and are word-aligned. The first byte is the length, the rest are the actual characters. This format applies even if the literal should be interpreted as a packed array of characters.
- All reals, sets, and long integers are word-aligned and in REVERSE word order, that is, the higher-order bits of the real or set are in lower-numbered memory locations.

Conventions and Notation

Each operand on the evaluation stack (for example, tos or tos−1), can contain from one byte to 256 bytes, depending on its type and value. Unless specifically noted to the contrary, operands used by an instruction are popped off the evaluation stack (removed from the stack and not returned) as they are used.

In the descriptions of the various P-machine instructions the parameters are given as UB , SB , DB , B , or W . The term tos means the operand on the top of the evaluation stack, tos−1 is the next operand, and so on. The columns of information in the various instruction descriptions have the following meanings:

Column 1	**Column 2**	**Column 3**	**Column 4**
op-code mnemonic	decimal op-code	instruction parameters	full name and operation of the instruction

One-Word Loads and Stores

Constant

SLDC_0 SLDC_1 : SLDC_127	0 1 : 127		Short load one-word constant. For an instruction SLDC_x , push the opcode, x , with the high byte zero. That is, push an integer with the value x.

LDCN	159		Load constant NIL. Push Ø.
LDCI	199	W	Load one-word constant. Push W.

Local

SLDL_1 SLDL_2 : : SLDL_16	216 217 : 231		Short load local word. For an instruction SLDL_x , fetch the word with offset x in the data area of the executing procedure's activation record and push it.
LDL	2Ø2	B	Load local word. Fetch the word with offset B in the data area of the executing procedure's activation record and push it.
LLA	198	B	Load local address. Fetch the address of the word with offset B in the data area in the executing procedure's activation record and push it.
STL	2Ø4	B	Store local word. Store tos into word with offset B in the data area of the executing procedure's activation record.

Global

SLDO_1 SLD_2 : : SLDO_16	232 233 : 247		Short load global word. For an instruction SLDO_x , fetch the word with offset x in the data area of the activation record of the base procedure and push it.

LDO	169	B	Load global word. Fetch the word with offset B in the data area of the activation record of the base procedure and push it.
LAO	165	B	Load global address. Fetch the address of the word with offset B in the data area of the activation record of the base procedure and push it.
SRO	171	B	Store global word. Store tos into the word with offset B in the data area of the activation record of the base procedure.

Intermediate

LOD	182	DB,B	Load intermediate word. Fetch the word with offset B in the activation record found by traversing DB links in the static chain, and push it.
LDA	178	DB,B	Load intermediate address. Fetch address of the word with offset B in the activation record found by traversing DB links in the static chain, and push it.
STR	184	DB,B	Store intermediate word. Store tos into the word with offset B in the activation record found by traversing DB links in the static chain.

Indirect

SIND_0	248		Load indirect word. Fetch the word pointed to by tos and push it (this is a special case of SIND_x , described below).
SIND_1 SIND_2 : : SIND_7	249 250 : 255		Short index and load word. For an instruction SIND_x , index the word pointer tos by x words, and push the word pointed to by the result.
IND	163	B	Static index and load word. Index the word pointer tos by B words, and push the word pointed to by the result.
STO	154		Store indirect word. Store tos into the word pointed to by tos−1 .

Extended

LDE	157	UB,B	Load extended word. Fetch the word with offset B in the data segment number UB (of an intrinsic unit) and push it.
LAE	167	UB,B	Load extended address. Fetch the address of the word with offset B in the data segment number UB (of an intrinsic unit), and push it.
STE	209	UB,B	Store extended word. Store tos into the word with offset B in the data segment number UB (of an intrinsic unit).

Multiple-Word Loads and Stores (Sets and Reals)

LDC	179	UB, ⟨data⟩	Load multiple-word constant. Fetch the word-aligned ⟨data⟩ of UB words in reverse word order, and push the data.
LDM	188	UB	Load multiple words. Fetch UB words of word-aligned data in reverse order, whose beginning is pointed to by tos , and push the block.
STM	189	UB	Store multiple words. Transfer UB words of word-aligned data in reverse order, whose beginnings pointed to by tos , to the location block pointed to by tos−1 .

Byte Array Handling

LDB	190		Load byte. Index the byte pointer tos−1 by the integer index tos , and push the byte (after zeroing high byte) pointed to by the resulting byte pointer.
STB	191		Store byte. Index the byte pointer tos−2 by the integer index tos−1 , and push the byte tos into the location pointed to by the resulting byte pointer.

String Handling

LSA	166	UB,⟨chars⟩	Load constant string address. Push a word pointer to the constant character string UB,⟨chars⟩ onto the evaluation stack. As the constant string is contained in the code segment, and may not be in the stack/heap space, a copy of the string is pushed onto the program stack. If this string has not previously been pushed onto the stack during the currently-active procedure, copy UB⟨chars⟩ onto the program stack (add one space to the end of the string if UB⟨chars⟩ is an even number of characters); push a 16-bit integer onto the program stack that points to the first byte of the string in the procedure code; push a 16-bit linkage pointer onto the program stack that points to the string or packed array most recently pushed onto the program stack (the linkage pointer is Ø if no other string or packed array has yet been pushed onto the stack); push a pointer onto the evaluation stack that points to the string length byte UB on the program stack.

			If UB⟨chars⟩ has been pushed onto the stack during the currently-active procedure, push a pointer onto the evaluation stack that points to the string length byte UB on the program stack. The contents of the program stack are not changed, which prevents needless, possibly stack-overflowing entries. In either case, advance the IPC register past the original copy of the string in the code space.
SAS	170	UB	String assign. tos is either a source byte pointer or a character. (Characters always have a high byte of zero, while pointers never do.) tos−1 is a destination byte pointer. UB is the declared size of the destination string. If the declared size is less than the current size of the source string, give an execution error; otherwise transfer all bytes of source containing valid information to the destination string.
IXS	155		Index string array. tos−1 is a byte pointer to a string. tos is an index into the string. Check to see that the index is in the range 1..current string length. If so, continue execution; if not, give an execution error.

Record and Array Handling

MOV	168	B	Move words. Transfer a source block of B words, pointed to by byte pointer tos, to a similar destination block pointed to by byte pointer tos−1.
INC	162	B	Increment field pointer. Index the word pointer tos by B words and push the resultant word pointer.
IXA	164	B	Index array. tos is an integer index, tos−1 is the array base word pointer, and B is the size (in words) of an array element. Compute a word pointer (tos−1) + (B * tos) to the indexed element and push the pointer.
IXP	192	UB1,UB2	Index packed array. tos is an integer index, tos−1 is the array base word pointer. UB1 is the number of elements per word, and UB2 is the field width (in bits). Compute a packed field pointer to the indexed field and push the resulting pointer.
LPA	208	UB,〈chars〉	Load a packed array. Push a word pointer to the packed array 〈chars〉 onto the evaluation stack. As the packed array is contained in the code segment, and may not be in the stack/heap space, a copy of the array is pushed onto the program stack. If this packed array has not

previously been pushed onto the stack during the currently-active procedure, copy ⟨chars⟩ onto the program stack (add one space to the end of the packed array if ⟨chars⟩ has an odd number of characters); push a 16-bit integer onto the program stack that points to the first byte of the packed array in the procedure code; push a 16-bit linkage pointer onto the program stack that points to the string or packed array most recently pushed onto the program stack (the linkage pointer is 0 if no other string or packed array has yet been pushed onto the stack); push a pointer onto the evaluation stack that points to the first byte of the packed array on the program stack.

If the same packed array has been pushed onto the stack during the currently-active procedure, push a pointer onto the evaluation stack that points to the first byte of the packed array on the program stack. The contents of the program stack are not changed, which prevents needless, possibly stack-overflowing entries.

In either case, advance the IPC register past the original copy of the string in the code space.

LDP	186		Load a packed field. Fetch the field indicated by the packed field pointer tos , and push it.
STP	187		Store into a packed field. Store the data tos into the field indicated by the packed field pointer tos−1 .

Dynamic Variable Allocation

Note that the NP, XNP register points to the current top of the heap (one byte beyond the last byte in use). GDIRP is a SYSCOM field that points to the top of a temporary directory buffer above the heap.

NEW	158	1	New variable allocation. tos is the size (in words) to allocate for the variable, and tos−1 is a word pointer to a pointer variable. If the GDIRP field is non-NIL, set GDIRP to NIL. Store the NP register into the word pointed to by tos−1 , and increment the NP register by tos words.
MRK	158	31	Mark heap. Set the GDIRP field to NIL, then store the NP register into the word indicated by the word pointer tos .
RLS	158	32	Release heap. Set the GDIRP field to NIL, then store the word indicated by the word pointer tos into the NP register.

Top-of-Stack Arithmetic

Integers

Note: Overflows do not cause an execution error; they are ignored and the results are undefined.

ABI	128	Absolute value of integer. Push the absolute value of the integer tos. The result is undefined if tos is initially −32768.
ADI	130	Add integers. Add tos and tos−1, and push the resulting sum.
NGI	145	Negate integer. Push the two's complement of tos. The result is undefined if tos is initially −32768.
SBI	149	Subtract integers. Subtract tos from tos−1, and push the resulting difference.
MPI	143	Multiply integers. Multiply tos and tos−1, and push the resulting product.
SQI	152	Square integer. Square tos, and push the result.
DVI	134	Divide integers. Divide tos−1 by tos and push the resulting integer quotient (any remainder is discarded). Division by zero causes an execution error.

MODI	142		Modulo integers. Divide tos−1 by tos and push the resulting remainder.
CHK	136		Check against subrange bounds. Insure that tos−1 <= tos−2 <= tos , leaving tos−2 on the stack. If conditions are not satisfied, give an execution error.
EQUI	195		tos−1 = tos .
NEQI	203		tos−1 <> tos .
LEQI	200		tos−1 <= tos .
LESI	201		tos−1 < tos .
GEQI	196		tos−1 >= tos .
GRTI	197		tos−1 > tos .
			Integer comparisons. Compare tos−1 to tos and push the result, TRUE or FALSE.

Non-Integer Comparisons

The next six instructions are non-specific non-integer comparisons. Comparisons using specific values of UB are given in later sections.

EQU	175	UB	tos−1 = tos .
NEQ	183	UB	tos−1 <> tos .
LEQ	180	UB	tos−1 <= tos .
LES	181	UB	tos−1 < tos .
GEQ	176	UB	tos−1 >= tos .

GRT 177 UB tos−1 > tos .
Compare tos−1 to tos , and push the result, TRUE or FALSE. The type of comparison is specified by UB :

Contents of tos − 1 & tos	**Value of UB for Comparison**
reals	2
strings	4
booleans	6
sets	8
byte arrays	10
words	12

Reals

FLT 138 Float top-of-stack. Convert the integer tos to a floating point number, and push the result.

FLO 137 Float next to top-of-stack. tos is a real, tos−1 is an integer. Convert tos−1 to a real number, and push the result.

TNC 158 22 Truncate real. Truncate (as defined in Jensen and Wirth*) the real tos and convert it to an integer, and push the result.

RND 158 23 Round real. Round (as defined in Jensen and Wirth*) the real tos , then truncate and convert to an integer, and finally push the result.

ABR 129 Absolute value of real. Push the absolute value of the real tos .

ADR	131	Add reals. Add tos and tos−1 , and push the resulting sum.
NGR	146	Negate real. Negate the real tos , and push the result.
SBR	150	Subtract reals. Subtract tos from tos−1 , and push the resulting difference.
MPR	144	Multiply reals. Multiply tos and tos−1 , and push the resulting product.
SQR	153	Square real. Square tos , and push the result.
DVR	135	Divide reals. Divide tos-1 by tos , and push the resulting quotient.
POT	158 35	Power of ten. If the integer tos is in the range 0 <= tos <= 38 , push the real value 10^tos . If the integer tos is not in this range, give an execution error.
EQUREAL	175 2	tos−1 = tos .
NEQREAL	183 2	tos−1 <> tos .
LEQREAL	180 2	tos−1 <= tos .
LESREAL	181 2	tos−1 < tos .
GEQREAL	176 2	tos−1 >= tos .
GTRREAL	177 2	tos−1 > tos .
		Real comparisons. Compare the real tos−1 to the real tos , and push the result, TRUE or FALSE.

* Kathleen Jensen and Niklaus Wirth: *Pascal User's Manual and Report*, 2nd ed. (New York: Springer-Verlag, 1978), p. 13

Strings

EQUSTR	175 4	tos−1 = tos .
NEQSTR	183 4	tos−1 <> tos .
LEQSTR	180 4	tos−1 <= tos .
LESSTR	181 4	tos−1 < tos .
GEQSTR	176 4	tos−1 >= tos .
GRTSTR	177 4	tos−1 > tos .

String comparisons. Find the string pointed to by word pointer tos−1 , compare it alphabetically to the string pointed to by word pointer tos , and push the result, TRUE or FALSE.

Logical

LAND	132	Logical AND. Push the result of tos−1 AND tos . This is a bitwise AND of two 16-bit words.
LOR	141	Logical OR. Push the result of tos−1 OR tos . This is a bitwise OR of two 16-bit words.
LNOT	147	Logical NOT. Push the one's complement of tos . This is a bitwise negation of one 16-bit word.

EQUBOOL	175 6	tos−1 = tos .
NEQBOOL	183 6	tos−1 <> tos .
LEQBOOL	180 6	tos−1 <= tos .
LESBOOL	181 6	tos−1 < tos .
GEQBOOL	176 6	tos−1 >= tos .
GRTBOOL	177 6	tos−1 > tos .

Boolean comparisons. Compare bit 0 of tos−1 to bit 0 of tos and push the result, TRUE or FALSE.

Sets

ADJ	160	UB	Adjust set. Force the set tos to occupy UB words, either by expansion (putting zeros "between" tos and tos−1) or by compression (chopping high words off the set), discard the length word, and push the resulting set.
SGS	151		Build a one member set. If the integer tos is in the range 0 <= tos <= 511 , push the set [tos] . If not, give an execution error.
SRS	148		Build a subrange set. If the integer tos is in the range 0 <= tos <= 511 , and the integer tos−1 is in the same range, push the set [tos−1..tos] (push the set [ ] if tos−1 > tos). If either integer exceeds the range, give an execution error.
INN	139		Set membership. If integer tos−1 is in set tos , push TRUE. If not, push FALSE.
UNI	156		Set union. Push the union of sets tos and tos−1 . (tos OR tos−1)
INT	140		Set intersection. Push the intersection of sets tos and tos−1 . (tos AND tos−1)
DIF	133		Set difference. Push the difference of sets tos−1 and tos . (tos−1 AND NOT tos).

EQUPOWR	175	8	tos−1 = tos .
NEQPOWR	183	8	tos−1 <> tos .
LEQPOWR	180	8	tos−1 <= (is a subset of) tos .
GEQPOWR	176	8	tos−1 >= (is a superset of) tos .

Set comparisons. Compare set tos−1 to the set tos , and push the result, TRUE or FALSE.

Byte Arrays

EQUBYT	175	10, B	tos−1 = tos .
NEQBYT	183	10, B	tos−1 <> tos .
LEQBYT	180	10, B	tos−1 <= tos .
LESBYT	181	10, B	tos−1 < tos .
GEQBYT	176	10, B	tos−1 >= tos .
GRTBYT	177	10, B	tos−1 > tos .

Byte array comparisons. Compare byte array tos−1 to byte array tos , and push the result, TRUE or FALSE. Note: <=, <, >=, and > must be used with packed arrays of characters only. B specifies the number of bytes to compare.

Records and Word Array Comparisons

EQUWORD	175	12, B	tos−1 = tos .
NEQWORD	183	12, B	tos−1 <> tos .

Word or multiword structure comparisons. Compare word structure tos−1 to word structure tos , and push the result, TRUE or FALSE. B gives the number of bytes to compare.

Jumps

The JTAB, XJTAB register points to the highest word of the attribute table in the currently-executing procedure. The IPC, XIPC register points to the next instruction to be executed in the currently-activating procedure.

UJP 185 SB Unconditional jump. SB is a jump offset. If this offset is non-negative (a jump less than 128 bytes forward), it is simply added to the IPC register. (A value of zero for the jump offset will make any jump a two-byte NOP.) If SB is negative (a jump backward or more than 127 bytes forward), then SB is used as a byte offset into the jump table within the attribute table pointed to by the JTAB register, and the IPC register is set to the byte address (JTAB[SB]) − contents of (JTAB[SB]) .

FJP 161 SB False jump. Jump (as described for UJP) if tos is FALSE.

XJP 172 W1,W2,<case table>,W3

Case jump. W1 is word-aligned and the minimum case selector of the case table. W2 is the maximum case selector. W3 is an unconditional jump offset past the case table. The case table is (W2 − W1 + 1) words long, and contains self-relative pointers.

If tos , the case selector expression, is not in the range W1..W2 , then point the IPC register at W3 . Otherwise, use (tos − W1) as an index into the case table, and set the IPC register to the byte address (casetable[tos − W1]) minus the contents of (casetable[tos − W1]), and continue execution.

Procedure and Function Calls

The general method of procedure/function invocation is:

1. Find the procedure code of the called procedure.
2. From the DATA SIZE and PARAMETER SIZE fields of the attribute table of the called procedure, determine the size (in bytes) of the needed activation record, and extend the program stack by that number of bytes.
3. Copy the number of bytes specified by the PARAMETER SIZE field from the top of the evaluation stack (tos) to the beginning of the space just allocated on the program stack. This passes parameters to the new procedure from its calling procedure.
4. Build a markstack, saving the SP, IPC, XIPC, SEG, JTAB, KP, STRP, MP, and a static link pointer (MSSTAT) to the most recent activation record of the procedure that is the lexical parent of the called procedure.
5. Calculate new values for the SP, IPC, XIPC, JTAB, XJTAB, MP, XMP, and if necessary, the SEG, and XSEG registers. Issue an execution error if the program stack overflows.
6. If the called procedure has a lexical level of −1 or Ø (in other words, it is a base procedure) save the value of the BASE register on the evaluation stack and then equate the BASE register with the MP register.
7. Save the value of the KP register on the program stack.

8. Save the value of the STRP register on the program stack.
9. Calculate a new value for the KP register to set it one word beyond the value of the STRP register.

CLP	206	UB	Call local procedure. Call procedure number UB , which is an immediate child of the currently executing procedure and in the same segment. The MSSTAT field (static link) of the markstack is set to the value of the old MP register.
CGP	207	UB	Call **global procedure**. Call procedure number UB , which is at lexical level 1 and in the same segment as the currently executing procedure. The MSSTAT field (static link) of the markstack is set to the value of the BASE register.
CIP	174	UB	Call intermediate procedure. Call procedure number UB in the same segment as the currently executing procedure. The MSSTAT field (static link) of the markstack is set by looking up the dynamic chain (MSDYN fields) until an activation record is found whose caller had a lexical level one less than the procedure being called. Use that activation record's MSSTAT field (static link) as the static link of the new markstack.
CBP	194	UB	Call base procedure. Call procedure number UB , which is at lexical level −1 or 0. The MSSTAT field (static link) of the markstack is set to the MSSTAT field in the activation record of the procedure pointed to by the BASE register.

			The value of the BASE register is saved on the evaluation stack, after which it is set to point to the MSSTAT field of the activation record just created.
CXP	2Ø5	UB1,UB2	Call EXTERNAL procedure. Call procedure number UB2 , in segment UB1 . Used to call any procedure not in the same segment as the calling procedure, including base procedures. If the desired segment is not already in memory, it is read from disk. Build an activation record. Calculate the static link for the markstack (if the called procedure has a lex level of −1 or Ø, set as in the CBP instruction; otherwise set as in the CIP instruction).
CSP	158	UB	Call standard procedure. Used to call standard procedures built into the P-machine.
RNP	173	DB	Return from non-base procedure. DB is the number of words that should be returned as a function value (Ø for procedures, 1 for non-real functions, and 2 for real functions). Copy DB words from the higher addresses of the current procedure's activation record, and push them onto the evaluation stack. Then copy the information in the current procedure's markstack fields into the psuedo-registers to restore the calling procedure's correct environment.

RBP	193	DB	Return from base procedure. Move the value of the BASE register saved on the evaluation stack by a CBP back into the BASE register, and then proceed as in the RNP instruction.
EXIT	158 4		Exit from procedure. tos is the procedure number, tos−1 is the segment number. First, set the MSIPC field to point to the exit code of the currently executing procedure. If the current procedure is not the one to exit from, change the MSIPC field of each markstack to point to the exit code of the procedure that invoked it, until the desired procedure is found. Then continue execution. If at any time the saved MSIPC field of the main body of the operating system is about to be changed, give an execution error.

System Support Procedures

See the *Apple III Pascal Programmer's Manual*, Volume 1 for a description of the Pascal language level interface to these functions.

Byte Array Procedures

FLC	158 10		Fillchar. tos is the source character. tos−1 is the number of bytes in the source character which are to be filled. tos−2 is a byte pointer

		to the first byte to be filled in the destination. Copy the character tos into tos−1 characters of the destination.
SCN	158 11	Scan. tos is a two-byte quantity (usually the default integer Ø) which is pushed, but later discarded without being used in this implementation. tos−1 is a byte pointer to the first character to be scanned. tos−2 is the character against which each scanned character of the array is to be checked. tos−3 is Ø if the check is for equality, or 1 if the check is for inequality. tos−4 specifies the maximum number of characters to be scanned (scan to the left if negative). If a character check yields TRUE, push the number of characters scanned (negative, if scanning to the left). If tos−4 characters are scanned before character check yields TRUE, push tos−4 .
MVL	158 Ø2	Moveleft. tos specifies the number of bytes to move. tos−1 is a byte pointer to the first destination byte. tos−2 is a byte pointer to the first source byte. Copy tos bytes from the source to the destination, proceeding from left to right through both source and destination.

MVR	158 ∅3		Moveright. tos specifies the number of bytes to move. tos−1 is a byte pointer to the first destination byte. tos−2 is a byte pointer to the first source byte. Copy tos bytes from the source to the destination, proceeding from right to left through both source and destination.

Compiler Procedures

BPT	213	B	Breakpoint. Not used (acts as a NOP).
TRS	158 ∅8		Treesearch. tos−2 is a byte pointer to the root of a binary tree. tos is a byte pointer to a location which contains the address of an eight-character name to be found or placed in the tree. Search the tree, looking for a record with the required name. On completion of the search, store the address of the last node visited, into the location pointed to by the byte pointer tos−1, and push the result of the search: ∅ if the last node was a record with the search name, 1 if the search name should be a new record, attached to the last tree node by the Right Link, −1 if the search name should be a new record, attached to the last tree node by the Left Link.

This is an assembly-language binary tree search used by the Compiler. It is fast, but does *not* do type checking on the parameters. The binary tree uses nodes of type

```
CTP = RECORD
        NAME: PACKED ARRAY [1..8]
              OF CHAR;
        LLINK, RLINK: ^CTP;
           .
           .
           .
        END;
```

IDS	158 Ø7	Idsearch. Used by the Compiler to parse reserved words and identifiers.

Miscellaneous

TIM	158 Ø9	Time. Pop two pointers to two integers, and place zero in both integers.
XIT	214	Exit the operating system. Do a **warm boot** of the system, as the operating system's H(alt command.
NOP	215	No operation. Sometimes used to reserve space in the code for later additions.

6

Programming Techniques

6
Programming Techniques

This chapter is a collection of useful techniques and hints to use while programming with the Apple III Pascal system. It is divided into two parts: the first deals with the Pascal language, and the second discusses assembly-language techniques.

Apple III Packing Algorithm

Simple types (INTEGER, BOOLEAN, and so forth) in UCSD Pascal have two standard sizes, depending on whether or not they are packed. These standard sizes are:

Type	Standard Unpacked Size	Standard Packed Size
Integer	one word (16 bits)	one word
Real	two words	two words
Char	one word	one byte (8 bits)
Boolean	one word	one bit
Subrange	one word	if smallest value >= Ø, then number of bits in largest value else one word
Scalar	one word	number of bits needed to represent the number of scalars in the scalar list
Long Integer	For form INTEGER[I]: (I+3) DIV 4 + 1 words	(I+3) DIV 4 + 1 words
Pointer	one word	one word
String	For string of max length N: (N+2) DIV 2 words	(N+2) DIV 2 words

Complex types, including RECORDS, ARRAYS, FILES, and SETS, always occupy a whole number of words whether they are packed or not. The number of words occupied depends on the internal structure given to the type.

Records

Each field that is a simple type is allocated a size as indicated above. If the record is a packed record, then the packed sizes are allocated. Tag fields, *if* they are associated with a named variable, occupy the same space as they would if they were ordinary fields. (Untagged variants occupy no space.) For example, the record below indicates the number of words allocated to each field.

```
PACKED RECORD
  NAME : STRING[2Ø];                  {11 words}
  SEX  : (MALE, FEMALE);              {1 bit}
  ID   : Ø..8191;                     {13 bits}
  MARRIED:BOOLEAN;                    {1 bit}
  CASE HASCHILDREN:BOOLEAN OF         {1 bit}
    TRUE:(NUMCHILDREN:INTEGER;        {1 word}
          OLDEST:INTEGER);            {1 word}   these overlay
    FALSE:(STERILE:BOOLEAN;           {1 bit}        the same
           CASE BOOLEAN OF            {Ø bits}          space
             TRUE:(BLOODTYPE:Ø..6))   {3 bits}
END;
```

In this case, the total record size is 14 words with the first 11 words going to the NAME field, the next word for the SEX, ID, MARRIED and HASCHILDREN fields, and the last two words either going to the NUMCHILDREN and OLDEST fields or to the STERILE and BLOODTYPE fields, depending on the value of the HASCHILDREN tagfield.

Since the allocation of fields is right to left within a word, the SEX, ID, MARRIED and HASCHILDREN fields are allocated within word 12 as follows:

SEX : bit Ø
ID : bits 1..13
MARRIED : bit 14
HASCHILDREN : bit 15

NUMCHILDREN and OLDEST are allocated words 13 and 14, respectively. However, if this case variant of the record had been declared as

```
...
CASE HASCHILDREN:BOOLEAN OF
  TRUE:(NUMCHILDREN, OLDEST:INTEGER);
...
```

then OLDEST would have been allocated word 13 and NUMCHILDREN word 14, since the compiler allocates fields backwards within a such a list. (This *backwards allocation* also applies to lists of variables in VAR declarations.)

If a field is packable, but there is not enough room in a given word for that field to fit, the entire field is moved to the beginning of the next word. This leaves some unused space in the first word. An example is

```
TYPE PART = PACKED RECORD
       PARTNUM:Ø..511;       {word 1, bits Ø..8}
       AMOUNT: INTEGER;      {word 2, all bits}
       ORDERQTY: 1..99;      {word 3, bits Ø..6}
     END;
```

In this example bits 9 through 15 of the first word go unused because the integer won't fit there. Also, note that bits 7 through 15 of the third word go unused, but since the record size must be a whole number of words, the total record size is exactly three words.

Accordingly, if PART is used as part of a larger record

```
     PARTSHEET = PACKED RECORD
       WHICHPART:PART;      {words 1..3}
       INITIAL: CHAR;       {word 4, bits Ø..7}
     END;
```

the record type PART is considered to be a three-word chunk, and although the INITIAL field would have fit into the third word of PART, it is not put there.

Arrays

For an array to be packed, the size of the array element must be eight bits or less. Arrays of records or other complex types are not packed. If the element size is eight bits or less, then each 16-bit word of the array gets the largest possible integral number of elements. In the array

```
PACKED ARRAY [-10..10] OF 0..7;
```

each word of the array contains five three-bit elements (with bit 15 of each word empty); the array contains a total of five words (21 divided by 5, rounded up). Array elements are allocated in increasing word order in memory and in increasing bit order within each word.

Note that the array declarations

```
PACKED ARRAY [1..10] OF PACKED ARRAY [1..2] OF BOOLEAN;
PACKED ARRAY [1..10, 1..2] OF BOOLEAN;
ARRAY [1..10] OF PACKED ARRAY [1..2] OF BOOLEAN;
```

are all equivalent, and that the "inner" array of booleans gets packed into one word (14 bits unused), while the "outer" array of arrays does not get packed (the size of its element is one word).

Sets

Packed or unpacked, a set occupies the number of bits equal to the largest element's ordinal value plus one and is rounded up to a whole number of words. For example,

```
TYPE
  A = SET OF 20..63;
  B = SET OF 40..64;
```

allocates four words for A and five words for B.

Files

All files, packed or unpacked, currently occupy at least 550 words that are distributed as follows:

256 words for the block buffer

256 words for the index block buffer

38 words for the File Information Block

Typed files occupy an additional amount of space equal to the size of the type for the file window. Files of type TEXT or INTERACTIVE occupy 551 words.

Pascal Language Techniques

This section includes discussions of efficient use of variable references, CASE statements, string and packed array constants, and SEGMENT procedures. A group of useful Compiler options are also discussed.

Dynamic Text Arrays

The following fragment of Pascal-code demonstrates a method by which you can dynamically allocate a variable-length packed array of characters (a text array). The procedure works in the following manner:

1. A check is made to ensure that there is enough space for the array. If there is not, a message is displayed, and the procedure is exited.
2. The number of bytes available for a dynamic buffer is calculated.
3. The first block of the buffer is allocated, and a pointer to its head is defined.
4. Other blocks are sequentially allocated until there is not enough space left to allocate another.
5. All of the blocks in the buffer are transformed into a packed array of characters.

```
PROCEDURE CreateArray;

CONST
   FreeSpace= 2000;            {Words of stack/heap space to be
                                reserved to prevent overflow}
   BytesInBlock= 511;          {Number of bytes in a block minus one}
   WordsInBlock= 256;          {Number of words in a block}
   BytesInArray= 8000;         {Maximum number of bytes in text array}
   MaxArrayIndex= 7999;        {Maximum index into text array}

{Note: the values assigned to BytesInArray and MaxArrayIndex can approach
32767, but are limited by program and memory size}

TYPE
   BlockBuffer= PACKED ARRAY [0..BytesInBlock] OF CHAR;
                               {The block-sized input/output buffer}
   TextArray= PACKED ARRAY [0..MaxArrayIndex] OF CHAR;
                               {The text array, divided into BlockBuffer-
                                sized chunks}

VAR
   Loop,
   WordsInArray,               {Maximum number of words in the array}
   BytesCalcBuffer,            {Number of bytes allowed in the buffer}
   WordsCalcBuffer,            {Number of words allowed in the buffer}
   BytesActualBuffer :         {Number of bytes currently in the buffer}
            INTEGER;
   PtrBuffer : ^BlockBuffer;   {Pointer to buffer}
   PtrArray : ^TextArray;      {Pointer to text array}
   TrixBuffer : PACKED RECORD  {Record for conversion of buffer to a
                                text array, and for use as a temporary
                                buffer pointer}
              CASE BOOLEAN OF
                TRUE: (IB: ^TextArray);
                FALSE: (BB: ^BlockBuffer);
             END;

BEGIN

   {Check to see if there is enough room to allocate the buffer
    for the array.  Note: MEMAVAIL returns the number of available
    words.}
   IF MEMAVAIL < Freespace THEN BEGIN
      WRITELN ('Not enough room for text buffer.');
      READLN;
      EXIT (CreateArray)
   END;
```

```
{Calculate the number of bytes allowed in the buffer; defined as
 the smaller of "available memory" or the defined array size
 (BytesInArray)}
WordsCalcBuffer := MEMAVAIL - Freespace;
WordsInArray:= (BytesInArray DIV 2);
   IF WordsCalcBuffer > WordsInArray THEN
      BytesCalcBuffer:= BytesInArray
   ELSE BytesCalcBuffer:= WordsCalcBuffer * 2;

{Allocate the space for the buffer}
   NEW (TrixBuffer.BB);         {Allocate the first block, with a
                                 pointer to its head}

   {Allocate the remaining blocks in the buffer.  Since the 2nd
    through nth blocks are allocated sequentially after the 1st
    block, their pointers are discarded.}
   FOR Loop:= 1 to (BytesCalcBuffer DIV WordsInBlock - 1) DO
      NEW (PtrBuffer);

{Transform the buffer into an array to enable byte-oriented procedures
 and functions}
   PtrArray:= TrixBuffer.IB;
   BytesActualBuffer:= BytesCalcBuffer;

END;
```

Once the text array has been created, you are free to use byte-oriented procedures and functions, such as SCAN and MOVELEFT, with PtrArray as a parameter. Individual characters within the array can be referenced as

```
PtrArray^[Element]
```

where Element is in the range 0..BytesCalcBuffer. If you attempt to write to elements outside of this range, you will probably overwrite part of your program.

Segment Procedures

There is a limit of 160 procedures per segment (no more than 149 of which can be P-code procedures). If you require a greater number of procedures within a segment, use nested SEGMENT procedures.

Variable Declarations

Declare the most-frequently referenced variables in the first 16 words of each procedure's data space. Referencing the first 16 words in the activation record of a procedure is faster and requires less code than does referencing other variables in the activation record, because special P-codes exist for references to the first 16 words.

String and Packed Array Constants

String and packed array constants are stored in a linked list on the program stack. Each time a given string or packed array constant is referenced, the linked list must be traversed until the desired constant is located. A lengthy linked list will decrease program performance; instead, setting variables to constant values will improve performance.

Case Statements

Avoid using CASE statements with widely spaced case selectors. To implement a CASE statement, the Compiler builds a table in the code with an entry for each possible case selector from the smallest actually used to the largest. For example,

```
CASE letter OF 'a','g','m','z'
```

will cause a table with 26 jump vectors to be built. Consider substituting nested IF..THEN..ELSE's in place of such CASE statements.

Private Files

The Compiler E+ option allows declaration of file variables in the IMPLEMENTATION part (and not just the INTERFACE part) of units. Files declared in the INTERFACE of a unit or in a VAR declaration of the IMPLEMENTATION of a unit are global, and a 1K byte I/O buffer is allocated as

long as the program is running. Files declared in procedure headings of the IMPLEMENTATION of a unit are private to the unit, and their 1 K byte I/O buffer is allocated only as long as the procedure is active. Declaring files in the procedure heading of the IMPLEMENTATION of units allows you to regain the 1K bytes of the I/O buffer when the procedure terminates.

Private files will only work with SOS-formatted disks, and not with Apple II UCSD format disks.

The IOCHECK and RANGECHECK Compiler Options

These compiler options provide runtime error checking. You can use I– and R– to defeat the checking, which will increase execution speed and decrease code size, at the expense of decreased automatic error checking.

The Resident Compiler Option

When there are no more active invocations of procedures in a segment, the segment code is removed from memory. Loading segments requires time and slows program execution. You can increase execution speed at the expense of additional memory by use of the RESIDENT Compiler option, which allows you to specify certain units and/or SEGMENT procedures to remain resident. In a computer with a large amount of memory (for example, 256K bytes) you can increase execution speed by keeping a large number of segments resident.

Residence Chains

The following fragment of Pascal code demonstrates a method for controlling the residence (and hence swapping) of segments in a Pascal program depending on the size of the system. The procedure INITCODE, not presented here, would presumably determine the size of available memory on the system on which the program is running. On systems beneath a certain

size, all segments except PERMAMENT would be swapped in and out under system control as indicated by the NOLOAD compiler option. For mid-sized configurations, the additional segments *A*, *B*, and *C* would also be permanently resident. On the largest configurations, *D*, *E*, and *F* would also be resident.

```
PROGRAM ExampleProgram;

USES  {$Using SOSIO.CODE}  SOSIO, {Uses SOS_IO memory management procedures
           in functional form to discover the memory available in the system}
     Permanent, A, B, C, D, E, F, G, H;

TYPE
      bbpp        =     PACKED RECORD    {bank/page (bb:pp) from SOS find_seg}
                          pp, bb: 0..255;
                        END;              {Used to find memory available}


VAR
   SwapAll, SwapSome : BOOLEAN;

   find_plist:         PACKED RECORD      {SOS find_seg param list}
                          pages: integer;
                          base: bbpp;
                          limit: bbpp;
                          segnum: integer;
                          rc: integer;
                       END;
.
.
.
PROCDURE Main;
   BEGIN
   .
   .
   .
   END;
```

```
PROCEDURE InitCode;
   VAR rc: Integer; {Return code from SOS_REL_SEG}

   BEGIN
   {Uses FIND_SEG in SOS_IO to determine memory available; sets SwapAll and
                SwapSome}
   SwapAll := FALSE;
   SwapSome := FALSE;

   WITH find_plist DO
      BEGIN
      pages := maxint; {try to get all we can}
      IF NOT sos_find_seg(2,    {may cross bank boundaries}
                         127, { $7F, a user segment type}
                         find_plist) THEN
         BEGIN
         {SOS_FIND_SEG returns FALSE if the number of pages originally
          requested cannot be allocated; the maximum number of pages available
          is placed in the pages field of FIND_PLIST in that case.  We call
          SOS_FIND_SEG again to be sure we can get it.}
         IF NOT sos_find_seg(2, 127, find_plist) THEN
            BEGIN
            pages := 0;
            END;
         END;
      {The pages field now contains the largest number of 256 byte pages
       available on the system.}
      IF pages > 10 {You must pick your own number!} THEN
         SwapAll := TRUE
      ELSE IF pages > 0 THEN
         SwapSome := TRUE;
      {Now release the segment}
      IF NOT sos_rel_seg(find_plist.segnum, rc) THEN
         writeln('Could not release segment (SOS release_seg error ', rc, ')');
      END;

   ...
   END;

PROCEDURE GetRest;
   BEGIN
     {$RESIDENT D,RESIDENT E,RESIDENT F} {Note that G and H are swapped on
                                          all systems.}
     Main;
   END;
```

```
PROCEDURE GetSome;
   BEGIN
      {$R A,R B,R C} {Note that R is an abbreviation for RESIDENT compiler
                      option.}
      IF SwapSome THEN
         BEGIN
         Main;
         END
      ELSE
         BEGIN
         GetRest;
         END;
   END;

BEGIN
   {$R Permanent,NOLOAD+} {Permanent is present on all systems; don't let
                           anything else be loaded unless some part active}

   InitCode; {Checks available memory; sets SwapAll and SwapSome.}

   IF SwapAll THEN
      BEGIN
      Main;
      END
   ELSE
      BEGIN
      GetSome;
      END;
END.
```

Pascal Unit Numbers and SOS Device Names and Numbers

The following Pascal program makes use of the functional version of SOS device calls available through the unit SOS_IO. It translates between SOS device names, SOS device numbers, and Pascal unit numbers.

```
{----- Start of Pascal Demonstration Program:  TestDevTranslation--------}
PROGRAM TestDevTranslation;

USES  {$Using SOSIO.CODE}  SOSIO; {Uses SOS_IO device calls in functional form}

VAR InString,                    {user input}
    SosName:STRING;              {Sos Name of device specified}
    PasNum,                      {Pascal Unit # of device specified}
    SosNum,                      {Sos device number of device specified}
    RetCode:INTEGER;             {Return code from SOS calls}
    Error: BOOLEAN;              {no device specified}
    DevList: PACKED ARRAY [0..10] OF 0..255;
                                 {Device information list returned by SOS}

FUNCTION GetSOSNum(PasNum:INTEGER):INTEGER;FORWARD;
FUNCTION GetPascalNum(SOSNum:INTEGER):INTEGER;FORWARD;


PROCEDURE Introduction;
BEGIN
   WRITELN('Welcome to the wonderful world of device translation!');
   WRITELN('Type in a device and I will translate it.');
   WRITELN('Formats are: SOS device number (e.g. 1) or Pascal unit(e.g. #4)');
   WRITELN('or even a SOS device name  (e.g. .rs232)');
   WRITELN;
   WRITELN('Type just a [RETURN] to exit');
   WRITELN;
   END;

FUNCTION GetSOSNum{PasNum:INTEGER):INTEGER};
   {returns SOS device number of unit numbered PasNum;
    0 if no such unit or no SOS device in that unit #}
   TYPE Byte = 0..255;
   VAR
      Data: PACKED RECORD
               RegularUnits: PACKED ARRAY [1..20] OF Byte;
               UserUnits:    PACKED ARRAY [128..147] OF Byte;
            END;
```

```
BEGIN
   UnitStatus(0,Data,0);                    {ask interpreter for table}
   IF PasNum IN [1..20] THEN
      BEGIN
      GetSOSNum := Data.RegularUnits[PasNum];
      END
   ELSE IF PasNum in [128..147] THEN
      BEGIN
      GetSOSNum := Data.UserUnits[PasNum];
      END
   ELSE
      BEGIN
      GetSOSNum := 0;
      END;
END;


FUNCTION GetPascalNum{SOSNum:INTEGER):INTEGER};
{returns the Pascal unit number of the SOS device numbered SOSNum;
 0 if none found }

TYPE Byte = 0..255;
VAR
   PasNum:INTEGER;
   Data: PACKED RECORD
            RegularUnits: PACKED ARRAY [1..20] OF Byte;
            UserUnits:    PACKED ARRAY [128..147] OF Byte;
         END;

 BEGIN
    IF SosNum = 0 THEN
       BEGIN
       PasNum := 0;   {avoid "holes" in Unittable}
       END
    ELSE
       BEGIN
       UnitStatus(0,Data,0);                    {ask interpreter for table}
       PasNum := SCAN(41,=CHR(SOSNum),Data)+1;     {find SOSNum in table}
       IF PasNum = 42 THEN
          BEGIN
          {scanned off end of table}
          PasNum := 0;
          END
       ELSE IF PasNum > 20 THEN
          BEGIN
          {adjust index appropriately}
          PasNum := PasNum+107;
          END;
       END;
    GetPascalNum := PasNum;
 END;
```

```
FUNCTION Number(S:STRING;VAR n:INTEGER):BOOLEAN;
   VAR i:INTEGER;
BEGIN
   NUMBER := FALSE;
   S := CONCAT(S,' ');
   n := 0;
   i := 1;
   WHILE S[i] IN ['0'..'9'] DO
      BEGIN
      Number := TRUE;
      n := n*10+ORD(S[i])-ORD('0');
      i := i+1;
      END;
END;

BEGIN
   Introduction;
   REPEAT
   READLN(InString);
   IF LENGTH(InString) >0 THEN
      BEGIN
      Error := FALSE;
      IF InString[1] = '.' THEN
         BEGIN
         {must be a SOS device name}
         SosName := Instring;
         IF NOT SOS_Get_D_Num(InString, SosNum, RetCode) THEN
            BEGIN
            Writeln('SOS error #', RetCode, ' from SOS_Get_D_Num');
            END;
         PasNum := GetPascalNum(SosNum);
         IF PasNum = 0 THEN SosName := '';
         END
      ELSE IF InString[1] = '#' THEN
         BEGIN
         {must be a Pascal unit number}
         Delete(InString,1,1); {remove # sign}
         IF Number(InString,PasNum) THEN
            BEGIN
            SosNum := GetSOSNum(PasNum);
            IF NOT SOS_D_Info(SosNum, SosName, DevList, RetCode) THEN
               BEGIN
               Writeln('SOS error #', RetCode, ' from SOS_D_Info');
               END;
            END
```

```
            ELSE
               BEGIN
               Error := TRUE;
               END;
            IF SOSNum = 0 THEN PasNum := 0;
            END
         ELSE IF Number(InString,SosNum) THEN
            BEGIN
            {must have typed a number}
            IF NOT SOS_D_Info(SosNum, SosName, DevList, RetCode) THEN
               BEGIN
               Writeln('SOS error #', RetCode, ' from SOS_D_Info');
               END;
            PasNum := GetPascalNum(SosNum);
            IF PasNum = 0 THEN SosNum := 0;
            END
         ELSE
            BEGIN
            Error := TRUE;
            END;
         {NOT Error => (SOSNum=0 <=> PasNum=0 <=> SosName='') }
         IF SosNum = 0 THEN Error := TRUE;
         IF Error THEN WRITE(CHR(7)) {Sound a bell}
         ELSE Writeln(SosName:16,' <=> ',SOSNum:2,' <=> #',PasNum);
      END;
   UNTIL InString= '';
END.
{------------------- End of TestDevTranslation ---------------------------}
```

Pascal Use of SOS Extended Memory

This section describes techniques that can be used to access extra memory available on the larger memory configurations of the Apple III. Before reading further, you should review the section System Memory Use in Chapter 3.

Apple III Pascal is upwardly compatible with Apple II Pascal. One of the constraints this imposes on the design of the Apple III system is the restriction to a data space of 64K bytes. Although the system uses memory outside this data space for SOS, drivers, graphics space, interpreter and code segments, this restriction still interferes with programs that handle large quantities of data.

However, by making use of SOS the Pascal programmer can gain direct access to the extra memory. The following assembly-language procedures

illustrate the techniques involved in accessing more data. These procedures ask SOS to allocate more space to the program, to allow the transfer of data to and from this space, and then eventually to deallocate the space. More elaborate storage allocation schemes may be built on top of this package. In the following simple example, after initialization a string is stored in the extra space, retrieved, and displayed.

The routines are presented in two parts. The first is assembly language that contains some useful macros, the procedure X_MOVELEFT (an expanded version of the Pascal MOVELEFT procedure to move bytes no matter which bank is the source or destination), and the function ADDRESS that returns the address of its variable. The second part is a Pascal program that demonstrates the required declarations and the use of the procedures.

```
;--------------- Extra Storage Space Assembly-Language Procedures
        .TITLE  "X_moveleft - Extended moveleft for moving bytes across banks"
        .NOPATCHLIST
        .NOMACROLIST
;    *---------------------------------------------------------------------*
;    |                                                                     |
;    |                         <<< X_moveleft >>>                          |
;    |                                                                     |
;    | Extended version of Pascal's moveleft for moving data across banks |
;    |                                                                     |
;    *---------------------------------------------------------------------*

; This module is the code for the procedure x_moveleft. X_moveleft is a
; generalized version of Pascal's moveleft procedure.  Functionally, it does
; the same thing, i.e., moves bytes, in ascending order, from a source to a
; destination. But unlike moveleft, x_moveleft can move the bytes no matter
; which bank they are in. It is designed to be used in Pascal programs which
; wish to use the rest of the bank space in larger machines.

; X_moveleft has the following Pascal declaration and call:

; PROCEDURE x_moveleft(src_bank, src_addr,
;                      dst_bank, dst_addr,
;                      pages, partial: integer);
```

```
; where: src_bank = bank number (0, 1, 2, ...) of the source. A special value
;                   of -1 means use Pascal's bank.
;        src_addr = address of 256 byte page in the source bank
;                   ($0000 to $7FFF).  This may be obtained using the
;                   function ADDRESS also supplied in this module and
;                   described below.
;        dst_bank = bank number of destination. A special value of -1 means
;                   use Pascal's bank.
;        dst_addr = page address in the destination bank ($0000 to $7FFF). As
;                   with src_addr, the ADDRESS function may be used to get the
;                   address of a Pascal variable.
;        pages    = number of whole 256-byte pages to move.
;        partial  = number of bytes in final (or only page) to move.

; X-moveleft will move pages*256+partial bytes from the source at the address
; src_bank:src_addr ($bb:xxxx) to the destination at address dst_bank:dst_addr.
; A -1 for a bank value means to use Pascal's bank. Thus the following two calls
; are functionally equivalent:

;   x_moveleft(-1, address(s), -1, address(d), 0, n) <==> moveleft(s, d, n)

; Segment and bank values for data may be obtained by the SOS request_seg or
; find_seg calls. They return segment addresses of the form $bb:pp, where the
; pp is in the range $20 to $9F for banked-switched or segment addressing.
; X_movebytes uses extended indirect addressing. Thus the pp value must be
; converted to a 4-byte address and offset by $2000 to produce addresses in the
; range $0000 to $FFFF. This could produce an address of the form $bb:00xx,
; i.e., a reference to a zero page. X_movebytes checks for this case and
; adjusts the bank and address values accordingly ($bb:00xx becomes $bb-1:80xx).
               .PAGE

; Also supplied in this module is the function ADDRESS:

; FUNCTION address(VAR x): integer;

; This returns the address of x as the value of the function. It is useful for
; moving Pascal data with X_MOVELEFT as illustrated above.



;
; Macro to push a word on to the stack.
;
               .MACRO PUSH
               LDA     %1+1
               PHA
               LDA     %1
               PHA
               .ENDM
```

```
;
; Macro to pop the stack into a word
;
                .MACRO POP
                PLA
                STA     %1
                PLA
                STA     %1+1
                .ENDM
                .PAGE

; The following macro saves SRC_BANK and DST_BANK (passed as the
; first parameter, %1, an extended address bank pointer) in SAVE_SRC_BANK
; and SAVE_DST_BANK, respectively, and then sets the new value from
; the stack (popping it off).  It turns the high bit on to enable indirect
; addressing.

; Follows convention that value of -1 means that the Pascal bank is to
; be used. It also checks the address currently pointed to by the corresponding
; word in zero page, and modifies it and %1 (bank register), if necessary, to
; make sure that the address does not point to the zero page of the bank pair
; (to avoid the hole in the memory map). Zero page wraparound during execution
; is taken care of by the main loop.
;
; This macro is commented as a pseudo procedure with the following declaration:
;
; PROCEDURE setbank(newbank: integer; VAR bank, temp: byte; addr: integer; );
;
; where: bank  = extended address bank pointer (%1, "src_bank" or "dst_bank")

;        newbank = new value for bank which is popped off the stack
;                  (from x_moveleft's call parameter list) by this macro.

;        temp    = place to save bank's former value before it is clobbered.
;                  Uses textual macro substitution to generate correct name.

;        addr    = a page address (src_addr or dst_addr) which is always
;                  $1601 below the bank register.  We look at the msb here,
;                  hence, we look at bank-$1601+1.
```

```
                .MACRO   SETBANK      ; PROCEDURE setbank(  newbank: integer;
                                      ;            VAR bank,temp: byte;
                                      ;            VAR addr: integer);
                LDA      %1           ;   BEGIN {setbank}
                STA      SAVE%1       ;     temp := bank; {save Pascal's value}
                PLA                   ;
                CMP      #0FF         ;     IF newbank<>-1 THEN
                BEQ      $1           ;        bank := newbank+$80; {high bit on}
                ORA      #80          ;
                STA      %1           ;
$1              LDA      %1-1601+1    ;     IF addr<256 THEN {have bank:00xx}
                BNE      $3           ;        BEGIN {make bank-1:80xx}
                LDA      #80          ;        addr := addr+$8000;
                STA      %1-1601+1    ;
                DEC      %1           ;        bank := bank-1;
$3              PLA                   ;        END;
                .ENDM                 ;   END; {setbank}
                .PAGE

; The following macro guarantees that the base pointer %1 (an address) will not
; wrap into zero page during next 256 increments of the pointer. If it would,
; then adjust the address to be in the first bank of a bank pair and increment
; the bank, i.e., bb:nnnn becomes bb+1:nnnn-$8000. Note that the corresponding
; bank register is at the address+$1601.

                .MACRO   CHKWRAP          ; PROCEDURE chkwrap(address: integer);
                LDA      %1+1             ;    BEGIN {chkwrap}
                CMP      #0FF             ;       IF address>=$FF00 THEN
                BCC      $1               ;          BEGIN {set to next bank}
                SBC      #80              ;          address := address-$8000;
                STA      %1+1             ;          bank := bank+1;
                INC      %1+1601          ;          END;
$1              .ENDM                     ;     END; {chkwrap}
                .PROC    X_MOVELEFT,6     ;

; PROCEDURE x_moveleft(src_bank, src_addr,
;                      dst_bank, dst_addr,
;                      pages, partial: integer);

; Move pages*256+partial bytes from the address src_bank:src_addr (extended
; indirect addressing form bb:xxxx) to address dst_bank:dst_addr. A bank of
; -1 means to use the Pascal bank.

SRC_ADDR        .EQU     0E0              ; page ptr to read bytes from
SRC_BANK        .EQU     1601+SRC_ADDR    ; X-byte for src_addr

DST_ADDR        .EQU     0E2              ; page ptr to write bytes to
```

```
DST_BANK        .EQU    1601+DST_ADDR  ; x_byte for dst_addr

RETURN          .EQU    0              ; return address
SAVE_SRC_BANK   .EQU    2              ; used to hold $16E1 across routine
SAVE_DST_BANK   .EQU    3              ; used to hold $16E3 across routine

COUNT_Z         .EQU    4              ; "Z"-byte of number of bytes to move
COUNT_Y         .EQU    5              ; "Y"-byte
COUNT_X         .EQU    6              ; "X"-byte (keep count bytes in order)


                                 ; PROCEDURE x_moveleft(src_bank,src_addr,
                                 ;                     dst_bank,dst_addr,
                POP     RETURN   ;                     pages,partial: integer);
                POP     COUNT_Z  ;  BEGIN {x_moveleft}
                PLA              ;    {set x,y,z values to reflect the
                CLC              ;     bytes to move: x*256*256+Y*256+z}
                ADC     COUNT_Y  ;    count_z := partial MOD 256;
                STA     COUNT_Y  ;    count_y := (pages MOD 256)+
                PLA              ;               (partial DIV 256);
                ADC     #0       ;    count_x := pages DIV 256;
                STA     COUNT_X  ;

                POP     DST_ADDR ;    {pop rest of the parameters}
                SETBANK DST_BANK ;    setbank(dst_addr,dst_bank,save_dst_bank);
                POP     SRC_ADDR ;
                SETBANK SRC_BANK ;    setbank(src_addr,src_bank,save_src_bank);

MOVE_PAGES                       ;    FOR i:=count_x downto 1 DO
                                 ;      BEGIN {move count_x*256 pages}
                CHKWRAP DST_ADDR ;      chkwrap(dst_addr);
                CHKWRAP SRC_ADDR ;      chkwrap(src_addr);
                LDA     COUNT_Y  ;      FOR j:=count_y DOWNTO 1 DO
                BNE     PAGE_LOOP ;       BEGIN {move count_y pages}
                LDA     COUNT_X  ;
                BEQ     PARTIAL  ;
                DEC     COUNT_X  ;
```

```
PAGE_LOOP      LDY     #0              ;  {move 1 page of 256 bytes}
$1             LDA     @SRC_ADDR,Y     ;  FOR y:=0 TO 255 DO
               STA     @DST_ADDR,Y     ;     dst_addr^[y] := src-addr^[y];
               INY                     ;
               BNE     $1              ;

               INC     SRC_ADDR+1      ;  src_addr := src_addr+256;
               INC     DST_ADDR+1      ;  dst_addr := dst_addr+256;
               DEC     COUNT_Y         ;  END; {moving count_y pages}
               JMP     MOVE_PAGES    ;  count_y := 256; {count_x*256 pages}
                                     ;  END; {moving count_x*256 pages}

PARTIAL        LDA     COUNT_Z   ;    {move remaining partial page}
               BEQ     EXIT      ;    FOR y:=0 TO count_z DO
               LDY     #0        ;      dst_addr^[y] := src_addr^[y];
$1             LDA     @SRC_ADDR,Y     ;
               STA     @DST_ADDR,Y     ;
               INY                     ;
               CPY     COUNT_Z         ;
               BNE     $1              ;

EXIT           LDA     SAVE_SRC_BANK ;{put things back the way they were}
               STA     SRC_BANK      ;src_bank := save_src_bank;
               LDA     SAVE_DST_BANK ;dst_bank := save_dst_bank;
               STA     DST_BANK        ;
               PUSH    RETURN          ;
               RTS             ;   END; {x_moveleft}

               .FUNC   ADDRESS,1       ;

; FUNCTION address(VAR x): integer;

;  Returns the address of "x".

RETURN         .EQU    0               ;

               POP     RETURN          ; FUNCTION address(VAR x): integer;
                                       ;    BEGIN {address}
               PLA                     ;
               PLA                     ;       {Remove the extra words on the}
               PLA                     ;       {stack because it's a function}
               PLA                     ;

               PUSH    RETURN          ;       address := tos; {param is still
                                       ;                        on top of stack}
               RTS                     ;    END; {address}

               .END
;--------------- End of Extra Storage Space Assembly-Language Procedures
```

```
[*-----------------------------------------------------------------------*
                         <<< USES_X_MOVELEFT >>>
 *-----------------------------------------------------------------------*]

PROGRAM uses_x_moveleft;

  USES {$USING SOSIO.CODE} sosio;  {This program uses the SOS_IO package in
               functional form to access the SOS memory management calls}

  {*-----------*
   | Constants |
   *-----------*}

   CONST

       MaxString = 100; {Maximum number of strings which can be entered in
                         the sample program}

  {*-------*
   | Types |
   *-------*}

  TYPE

     bbpp          =
                     PACKED RECORD    {bank/page (bb:pp) from find_seg}
                        pp, bb: 0..255;
                     END;
  {*-------------*
   | Global Data |
   *-------------*}

  VAR
     i, NumStrings:    0..MaxString;

     StringLoc: array [1..MaxString] OF RECORD
                                          bank, addr: integer;
                                        END;

     s: string[255];
```

```
{*----------------------------------------*
 | Data Used for SOS Segment Manipulations |
 *----------------------------------------*}

find_plist:
              PACKED RECORD      {find_seg param list}
                 pages: integer;
                 base: bbpp;
                 limit: bbpp;
                 segnum: integer;
                 rc: integer;
              END;

base_bank:    integer;           {segment starts in this bank}
base_addr:    integer;           {segment starts at this address}
limit_bank:   integer;           {segment ends in this bank}
limit_addr:   integer;           {segment ends at this address}

free_bank:    integer;           {free space in this bank}
free_addr:    integer;           {free space at this address}


{*---------------------------------*
 | External (assembly) Procedures |
 *---------------------------------*}

PROCEDURE x_moveleft(src_bank, src_addr, dst_bank, dst_addr, pages,
                     partial: integer);
   EXTERNAL ;

FUNCTION address(VAR x): integer;
   EXTERNAL ;

{*-------------------------------------*
 | alloc_segment - allocate a segment |
 *-------------------------------------*}

FUNCTION alloc_segment: integer;

   BEGIN {alloc_segment}
      WITH find_plist DO
         BEGIN
         pages := maxint; {try to get all we can}

         IF NOT sos_find_seg(2,    {may cross bank boundaries}
                    127, { $7F, a user segment type}
                    find_plist) THEN
```

```
            BEGIN
            {SOS_FIND_SEG returns FALSE if the number of pages originally
             requested cannot be allocated; the maximum number of pages
             available is placed in the pages field of FIND_PLIST in that
             case.  We call SOS_FIND_SEG again to be sure we can get it.}
            IF NOT sos_find_seg(2, 127, find_plist) THEN
               BEGIN
               writeln(
                  'Cannot allocate segment (SOS find_seg error ', rc, ')');
               pages := 0;
               END;
            END;

         base_bank := base.bb;
         base_addr := (base.pp-32)*256;
         limit_bank := limit.bb;
         limit_addr := (limit.pp-32)*256-1;

         free_bank := base_bank;
         free_addr := base_addr;

         alloc_segment := pages;
         END; {WITH}
   END; {alloc_segment}


{*---------------------------------------------*
 | free_segment - free the allocated segment |
 *---------------------------------------------*}

PROCEDURE free_segment;

   VAR
      rc: integer;

   BEGIN {free_segment}
      IF NOT sos_rel_seg(find_plist.segnum, rc) THEN
         writeln('Could not release segment (SOS release_seg error ', rc, ')');
   END; {free_segment}
{*----------------------------------------*
 | alloc - allocate space in the segment |
 *----------------------------------------*}

PROCEDURE alloc(amount: integer {nbr of bytes to allocate} ;
                VAR bank, addr: integer {location of allocated space} );

   VAR
      remain: integer;
      top_of_current_bank: integer;
```

```
   BEGIN {alloc - set bank to -1 if can't get the space}
      bank := free_bank;
      addr := free_addr;
      IF free_bank=limit_bank THEN
         BEGIN
         top_of_current_bank := limit_addr;
         END
      ELSE top_of_current_bank := 32767;
      remain := top_of_current_bank-free_addr+1;
      IF amount>remain THEN
         BEGIN
         IF free_bank<limit_bank THEN
            BEGIN
            free_bank := free_bank+1;
            free_addr := amount-remain;
            END
         ELSE
            BEGIN
            bank := -1;
            END;
         END
      ELSE
         BEGIN
         free_addr := free_addr + amount;
         END;
   END; {alloc}

{*------------------------------------------*
 | putstring - insert string into storage |
 *------------------------------------------*}
PROCEDURE putstring( image: string; VAR bank, addr: integer );

   VAR image_addr: integer; { address(image) }
       string_len: integer; { number of bytes of string plus 1 }

   BEGIN
   image_addr := address(image);

   string_len := length(image)+1; {include length byte}

   {allocate space in the extra space for the string}
      alloc(string_len, bank, addr);

   IF bank<0 THEN
      BEGIN {no more segment space}
      writeln('No space available for string');
      END {no segment space}
   ELSE {Move the string to the extra memory from the Pascal bank}
      x_moveleft(-1, image_addr, bank, addr, 0, string_len);

   END; {putstring}
```

```
{*------------------------------------------*
 | getstring - retrieve string from storage |
 *------------------------------------------*}
PROCEDURE getstring( VAR image: string; bank, addr: integer );

   VAR image_addr: integer; { address(image) }

   BEGIN

   { Read the string back }
      image_addr := address(image);
      x_moveleft(bank, addr, -1, image_addr, 0, 256);

      {We read back 256 bytes since length byte is stored away.  We thus can
      read back any string.}


   END; {getstring}


{*-----------------------*
 | init - initialization |
 *-----------------------*}

PROCEDURE init;

   VAR
      pages: integer;

   BEGIN {init}
      writeln(chr(28)); {Clear viewport}
      writeln;
      write('USES_X_MOVELEFT');
      writeln;


      {Get maximum number of 256 byte pages from SOS}
      pages := alloc_segment;

      {Terminate if number of pages is less than 25, an arbitrary number
       in this case.}
      IF pages<25 THEN
         BEGIN
         IF pages>0 THEN free_segment;
         writeln('Program terminated due to insufficient memory');
         exit(PROGRAM);
         END;
      writeln( 'Maximum storage space available is ', pages, ' pages.');
      writeln;

   END; {init}
```

```
{*-------------*
 | Main Program|
 *-------------*}


BEGIN {main program}
   init;

   NumStrings := 0;

   REPEAT
      {Read in a number of strings into the extra space; build
       an array of addresses}

      writeln('Enter a string.  Blank line terminates.');
      readln(s);
      NumStrings := NumStrings+1;
      putstring (s, StringLoc[NumStrings].bank, StringLoc[NumStrings].addr)

   UNTIL (s='') OR (NumStrings=MaxString) OR (StringLoc[NumStrings].bank<0 );

   NumStrings := NumStrings - 1; {Last is invalid: blank or no more room}

   writeln('Now we write them back out:');
   writeln;

   FOR i := 1 TO NumStrings DO
      BEGIN
      getstring (s, StringLoc[i].bank, StringLoc[i].addr);
      writeln(s);
      END;

   writeln;

   writeln('And that''s all of them!');

   free_segment;

END {USES_X_MOVELEFT} .
```

Assembly-Language Techniques

This section includes a hint on memory usage by segments with assembly-language procedures, and a list of useful assembly-language macros.

Assembly-Language Procedures

Segments with assembly-language procedures cannot be loaded across Apple III bank boundaries. To avoid this, the interpreter automatically selects the load location for a segment. Therefore, you should avoid placing assembly-language procedures in large segments. You could conceivably lose over 15K bytes of memory if a 16K segment with an assembly-language procedure were loaded in the stack/heap space such that it had to be pushed up above a bank boundary in the middle of this space. Instead, assembly-language procedures should be placed in their own segment and be USEd by the host program.

Macro Directives

This section consists of a group of macro directives that you may find useful in your assembly-language programs. Note: In the *Form* specification of each macro, parameters enclosed within () are required, while those enclosed within ⟨⟩ are optional.

The SOS Macro

This macro calls the specified SOS service using SOSBLK, a fixed area, as the SOS parameter buffer.

Form: SOS (service)

service: A SOS call number.

```
.MACRO      SOS
BRK
.BYTE       %1
.WORD       SOSBLK
.ENDM
```

The SOSCALL Macro

This macro calls the specified SOS service using a user-specified parameter buffer.

Form: SOSCALL (service),(pointer)

service: A SOS call number.

pointer: A SOS parameter block pointer.

```
.MACRO      SOSCALL
BRK
.BYTE       %1
.WORD       %2
.ENDM
```

The POP Macro

This macro saves the word on the top of the stack in a specified location; its action is complementary to the PUSH macro.

Form: POP (location)

location: The address in which the word is to be stored.

```
.MACRO      POP
PLA
STA         %1
PLA
STA         %1+1
.ENDM
```

The PUSH Macro

This macro pushes the word in a specified location onto the top of the stack; its action is complementary to the POP macro.

Form: PUSH (location)

location: The address from which the word is to be taken.

```
.MACRO    PUSH
LDA       %1+1
PHA
LDA       %1
PHA
.ENDM
```

The RMVBIAS Macro

This macro removes from the evaluation stack the four zero bytes (the bias) passed for a Pascal function.

Form: RMVBIAS

```
.MACRO
RMVBIAS
PLA
PLA
PLA
PLA
.ENDM
```

The MOVE Macro

This macro moves the word value stored at one location to another location.

Form: MOVE (from), (to)

from: The address whose value is to be moved.

to: The address to which the value is to be moved.

```
.MACRO    MOVE
LDA       %1
STA       %2
LDA       %1+1
STA       %2+1
.ENDM
```

The DEBUGSTR Macro

This macro generates ASCII strings to aid debugging, if DEBUG = 1 (TRUE). If DEBUG = Ø (FALSE), no strings are generated.

Form: DEBUGSTR (message),⟨jumpto⟩

message: The message to be inserted into the code as a .ASCII directive. Note that four asterisks are added before and after the message.

jumpto: The optional location to which execution should jump (to bypass the debug message).

```
.MACRO      DEBUGSTR
.IF         DEBUG
.IF         ''%2'' <> ''''
JMP         %2
.ENDC
.ASCII      ''*****%1*****''
.ENDC
.ENDM
```

The LOCALREG Macro

This macro initializes (zeros) the X-bytes zero-page address-pointer registers to access the currently switched-in memory bank. One to four zero-page addresses may be specified. The A register is destroyed; X and Y remain unchanged.

Form: PASCALRG (reg1),⟨reg2⟩,⟨reg3⟩,⟨reg4⟩

reg1, reg2, reg3, reg4: The locations of the zero-page registers to be initialized. Only reg1 is required.

```
          .MACRO     LOCALREG
XPGSTART  .EQU       1600
          LDA        #0
          STA        XPGSTART+1+%1
          .IF ''%4''<>''''
          STA        XPGSTART+1+%2
          STA        XPGSTART+1+%3
          STA        XPGSTART+1+%4
          .ELSE
          .IF ''%3''<>''''
          STA        XPGSTART+1+%2
          STA        XPGSTART+1+%3
          .ELSE
          .IF ''%2''<>''''
          STA        XPGSTART+1+%2
          .ENDC
          .ENDC
          .ENDC
          .ENDM
```

The PASCALRG Macro

This macro initializes the X-bytes of zero-page address-pointer registers to access a specific memory bank (in other words, enable enhanced indirect addressing). One to four zero-page addresses may be specified. The A register is destroyed; X and Y remain unchanged.

Form: PASCALRG (reg1),<reg2>,<reg3>,<reg4>

reg1, reg2, reg3, reg4: These are the locations of the zero-page registers to be initialized. Only reg1 is required.

INITXPG contains the value to which the registers should be initialized.

```
          .MACRO     PASCALRG
INITXPG   .EQU       16EF
XPGSTART  .EQU       1600
          LDA        INITXPG
          STA        XPGSTART+1+%1
          .IF ''%4''<>''''
          STA        XPGSTART+1+%2
          STA        XPGSTART+1+%3
          STA        XPGSTART+1+%4
```

```
•
.ELSE
.IF ''%3''<>''''
STA         XPGSTART+1+%2
STA         XPGSTART+1+%3
.ELSE
.IF ''%2''<>''''
STA         XPGSTART+1+%2
.ENDC
.ENDC
.ENDC
.ENDM
```

The SAVEREGS Macro

This macro saves the values of specified registers starting at a specific zero-page location. Any combination of A, X, and Y may be saved.

Form: SAVEREGS (location),⟨reg1⟩,⟨reg2⟩,⟨reg3⟩

location: The zero-page location at which the register values are to be saved. If this parameter is omitted, the `ZPAGE LOCATION NOT SPECIFIED` message will be displayed.

reg1, reg2, reg3: The registers from which the values are to be saved; they are all optional.

```
.MACRO      SAVEREGS
.IF         ''%1''=''''
ZPAGE LOCATION NOT SPECIFIED
.ENDC
.IF         ''%2''<>''''      ;Check for first register
.IF         ''%2''=''A''      ;Accumulator?
STA         %1
.ELSE
.IF         ''%2''=''X''      ;X register?
STX         %1
.ELSE
.IF         ''%2''=''Y''      ;Y register?
STY         %1
.ENDC
.ENDC
.ENDC
.ENDC
.IF         ''%3''<>''''      ;Check for second register
.IF         ''%3''=''A''      ;Accumulator?
STA         %1+1
```

```
.ELSE
.IF         ''%3''=''X''      ;X register?
STX         %1+1
.ELSE
.IF         ''%3''=''Y''      ;Y register?
STY         %1+1
.ENDC
.ENDC
.ENDC
.ENDC
.IF         ''%4''<>''''      ;Check for third register
.IF         ''%4''=''A''      ;Accumulator?
STA         %1+2
.ELSE
.IF         ''%4''=''X''      ;X register?
STX         %1+2
.ELSE
.IF         ''%4''=''Y''      ;Y register?
STY         %1+2
.ENDC
.ENDC
.ENDC
.ENDC
.ENDM
```

The RESTREGS Macro

This macro restores the values of specified registers, by reading values starting from a specific zero-page location. Any combination of A, X, and Y may be restored.

Form: RESTREGS (location),⟨reg1⟩,⟨reg2⟩,⟨reg3⟩

location: The zero-page location at which the register values are stored. If this parameter is omitted, the `ZPAGE LOCATION NOT SPECIFIED` message will be displayed.

reg1, reg2, reg3: The registers into which the values are to be restored; they are all optional.

```
.MACRO      RESTREGS
.IF         ''%1''=''''
ZPAGE LOCATION NOT SPECIFIED
.ENDC
```

```
.IF        ''%2''<>''''       ;Check for first register
.IF        ''%2''=''A''       ;Accumulator?
LDA        %1
.ELSE
.IF        ''%2''=''X''       ;X register?
LDX        %1
.ELSE
.IF        ''%2''=''Y''       ;Y register?
LDY        %1
.ENDC
.ENDC
.ENDC
.ENDC
.IF        ''%3''<>''''       ;Check for second register
.IF        ''%3''=''A''       ;Accumulator?
LDA        %1+1
.ELSE
.IF        ''%3''=''X''       ;X register?
LDX        %1+1
.ELSE
.IF        ''%3''=''Y''       ;Y register?
LDY        %1+1
.ENDC
.ENDC
.ENDC
.ENDC
.IF        ''%4''<>''''       ;Check for third register
.IF        ''%4''=''A''       ;Accumulator?
LDA        %1+2
.ELSE
.IF        ''%4''=''X''       ;X register?
LDX        %1+2
.ELSE
.IF        ''%4''=''Y''       ;Y register?
LDY        %1+2
.ENDC
.ENDC
.ENDC
.ENDC
.ENDM
```

The SET Macro

This macro sets specific bits within a byte.

Form: SET (bits),(byte)

bits: The bits to be set.

byte: The address of the byte whose bits are to be set.

```
        .MACRO    SET
        LDA       #%1
        ORA       %2
        STA       %2
        .ENDM
```

The RESET Macro

This macro resets specific bits within a byte.

Form: RESET (bits),(byte)

bits: The bits to be reset (set to Ø).

byte: The address of the byte whose bits are to be reset.

```
        .MACRO    RESET
MASK    .EQU      FF
        LDA       #%1∧MASK       ;∧ is EXCLUSIVE OR
        AND       %2
        STA       %2
        .ENDM
```

The SWITCH Macro

This macro performs an n-way branch based on a switch index. The maximum value of the switch index is 127 with bounds checking provided as an option. The A and Y registers, and the C, Z, and N status flags, are destroyed by the macro. The X register is *not* modified by the macro.

Form: SWITCH 〈index〉,〈bounds〉,(address table),〈*〉

index: The variable that is to be used as the switch index. If it is omitted, the accumulator is used as the index.

bounds: The maximum allowable value for the index. If the index exceeds this value, the carry bit is set and execution continues. If this parameter is omitted, then no bounds checking is performed.

address table: A table of addresses used by the switch. Note that the address−1 is used. This is because of the RTS instruction.

*: If the asterisk is supplied as the fourth parameter, the macro will push the switch address but will not exit to it; execution will continue after the macro.

```
        .MACRO      SWITCH
        .IF         ''%1''<>''''        ;If param1 then
        LDA         %1                  ;Load A with index
        .ENDC
        .IF         ''%2''<>''''        ;If param2 then
        CMP         #%2+1               ;Perform bounds check
        BCS         $Ø99                ;on switch index
        .ENDC
        ASL         A
        TAY
        LDA         %3+1,Y              ;Get switch address from the
        PHA                             ;table and push onto stack
        LDA         %3,Y
        PHA
        .IF         ''%4''<>''*''       ;If param4 <> * then
        RTS                             ;Exit to code
        .ENDC                           ;Else Continue
$Ø99
        .ENDM
```

The MOVEDATA Macro

This macro moves up to 255 bytes within the assembly-language code/data space, in descending order. The A and X registers are destroyed; Y is not modified.

Form: MOVEDATA (from), (to), (count)

from: The byte address of the location from which the move is to occur.

to: The byte address of the location to which the move is to occur.

count: The number of bytes to move. If count is zero, the message `ZERO IS A BAD COUNT` is displayed.

```
        .MACRO      MOVEDATA
        .IF         %3=0
        ZERO IS A BAD COUNT
        .ENDC
        LDX         #%3
$99     LDA         %1-1,X
        STA         %2-1,X
        DEX
        BNE         $99             ;Loop until done
        .ENDM
```

The MOVEDINC Macro

This macro moves up to 255 bytes within the assembly-language code/data space, in ascending order. The A and X registers are destroyed; Y is not modified.

Form: MOVEDINC (from), (to), (count)

from: The byte address of the location from which the move is to occur.

to: The byte address of the location to which the move is to occur.

count: The number of bytes to move. If count is zero, the message `ZERO IS A BAD COUNT` is displayed.

```
        .MACRO      MOVEDINC
        .IF         %3=0
        ZERO IS A BAD COUNT
        .ENDC
        LDX         #0
$99     LDA         %1,X
        STA         %2,X
```

```
        INX
        CPX         #%3
        BCC         $99                 ;Loop until done
        .ENDM
```

The BITBRANCH Macro

This macro causes a branch if specified bits within a byte are on or off. The A register is destroyed; X and Y are unmodified.

Form: BITBRNCH (data),<bitson>,<bitsoff>,(branch)

data: The location of the byte whose bits are to be checked.

bitson: The bits of this optional byte specify which bits of the data byte must be on if the branch is to occur.

bitsoff: The bits of this optional byte specify which bits of the data byte must be off if the branch is to occur.

branch: The address to which execution should branch if the bits of the data byte specified by bitson are on, and the bits specified by bitsoff are off.

If the bits specified by bitson are not on, or if the bits specified by bitsoff are not off, the specified branch is not taken. You need not specify both bitson and bitsoff, but you must specify at least one of them, or the message `NO BITS SPECIFIED` will be displayed.

```
        .MACRO      BITBRNCH
        .IF         ''%2''=''''
        .IF         ''%3''=''''
        NO BITS SPECIFIED               ;Generate an error
        .ELSE
        LDA         #%3
        AND         %1
        BEQ         %4                  ;Bits off only
        .ENDC
        .ELSE
        LDA         #%2
        AND         %1
        EOR         #%2
        .IF         ''%3''=''''
        BEQ         %4                  ;Bits on only
```

```
        .ELSE
        BNE     $Ø99
        LDA     #%3
        AND     %1
        BEQ     %4              ;Both conditions have been met
$Ø99
        .ENDC
        .ENDC
        .ENDM
```

The NOTBITBR Macro

This macro is the converse of macro BITBRNCH. It causes a branch if specified bits within a byte are not on or off. The A register is destroyed; X and Y are unmodified.

Form: NOBITBR (data),⟨bitson⟩,⟨bitsoff⟩,(branch)

data: The location of the byte whose bits are to be checked.

bitson: The bits of this optional byte specify which bits of the data byte must be on if the branch is not to occur.

bitsoff: The bits of this optional byte specify which bits of the data byte must be off if the branch is not to occur.

branch: The address to which execution should branch if the bits of the data byte specified by bitson are not all on, and the bits specified by bitsoff are not all off.

If any one of the bits specified by bitson are not on, or if any one of the bits specified by bitsoff are not off, the specified branch is taken. If the bits specified by bitson are on, and the bits specified by bitsoff are off, the specified branch is not taken, and execution continues with the next instruction. You need not specify both bitson and bitsoff, but you must specify at least one of them, or the message `NO BITS SPECIFIED` will be displayed.

```
        .MACRO      NOTBITBR
        .IF         ''%2''=''''
        .IF         ''%3''=''''
        NO BITS SPECIFIED       ;Generate an error
```

```
.ELSE
LDA      #%3
AND      %1
BNE      %4            ;Bits off only
.ENDC
.ELSE
LDA      #%2
AND      %1
EOR      #%2
.IF      ''%3''=''''
BNE      %4            ;Bits on only
.ELSE
BNE      %4
LDA      #%3
AND      %1
BNE      %4            ;Both conditions have been met
.ENDC
.ENDC
.ENDM
```

Equates for SOS Call Numbers

```
REQUEST_SEG     .EQU     040
FIND_SEG        .EQU     041
CHANGE_SEG      .EQU     042
GET_SEG_INFO    .EQU     043
GET_SEG_NUM     .EQU     044
RELEASE_SEG     .EQU     045
SET_FENCE       .EQU     060
GET_FENCE       .EQU     061
SET_TIME        .EQU     062
GET_TIME        .EQU     063
GET_ANALOG      .EQU     064
TERMINATE       .EQU     065
D_STATUS        .EQU     082
D_CONTROL       .EQU     083
GET_DEV_NUM     .EQU     084
D_INFO          .EQU     085
CREATE          .EQU     0C0
DESTROY         .EQU     0C1
RENAME          .EQU     0C2
SET_FILE_INFO   .EQU     0C3
```

```
GET_FILE_INFO   .EQU   0C4
VOLUME          .EQU   0C5
SET_PREFIX      .EQU   0C6
GET_PREFIX      .EQU   0C7
OPEN            .EQU   0C8
NEWLINE         .EQU   0C9
READ            .EQU   0CA
WRITE           .EQU   0CB
CLOSE           .EQU   0CC
FLUSH           .EQU   0CD
SET_MARK        .EQU   0CE
GET_MARK        .EQU   0CF
SET_EOF         .EQU   0D0
GET_EOF         .EQU   0D1
SET_LEVEL       .EQU   0D2
GET_LEVEL       .EQU   0D3
```

Glossary

The definitions given in this glossary are only those stated or implied in the text of this manual. Other definitions connected with different usages of the same terms are not given. An item appearing in the glossary is shown in boldface type when it first occurs in the text.

activation record memory space on the program stack that stores the markstack, function value, passed parameters, and local variables for an active procedure. Activation records are created by procedure calls and removed as a procedure is terminated.

Assembler directive statements placed in assembly-language programs that cause certain operations to be performed during program assembly. Assembler directives begin with a period, for example, .PROC .

attribute table a table associated with each procedure that contains information needed to execute the procedure. Attribute tables grow toward lower addresses.

automatic variable a variable for which space is allocated at the time the procedure declaring the variable is called.

bank a unit of memory of 32768 contiguous bytes.

BASE BASE procedure pointer. A 16-bit pointer on zero page that points to the MSSTAT field of the activation record of the most recently invoked base procedure. See **XBASE**.

base procedure a procedure of the Pascal system at lexical level Ø or −1.

base-relative relocation table a table of addresses, within an assembly-language procedure, each address to be relocated relative to the address contained in the BASE psuedo-register.

big a P-machine instruction parameter that is one-byte long when used to represent values in the range 0 through 127, and two-bytes long when used to represent values in the range 128 through 32767.

BIOS the Basic I/O System of the interpreter; it handles all low level Pascal I/O.

block a unit of storage of 512 contiguous bytes.

block boundary the boundary between byte 511 of one block and byte 0 of the next block.

byte eight bits of data.

byte-aligned an instruction or structure starting at any byte, not necessarily an even-numbered byte (see **word-aligned**).

codefile a file containing a segment dictionary and code segments.

code part a portion of a code segment that consists of a group of procedures together with descriptive information about the procedures (the procedure dictionary).

code segment a portion of a codefile containing P-code and/or native code. Code segments may have three parts: interface text, code part, and Linker information.

Compiler COMMENT option a Compiler option that allows you to specify a comment to be placed in the segment dictionary of a codefile.

data area the upper addresses of an activation record that contain space for local variables, passed parameters, and returned function value of a procedure.

data segment a portion of memory set aside at execution time as storage space for data of intrinsic units. In disk codefiles, data segments are simply an entry in the segment dictionary, as they have no interface text, code part, or Linker information.

declaration a Pascal construct that is used to announce the attributes of an identifier.

device a piece of hardware used for data input or output. A disk drive, video screen, and speaker are all commonly-used Apple III devices.

device driver the software interface to a device that enables the Apple III to communicate with that device.

don't-care byte Represents a non-negative integer less than 128; thus it can be treated as SB (signed byte) or UB (unsigned byte).

dynamic chain a series of dynamic links. The dynamic chain describes the "route" by which a procedure was called.

dynamic link a pointer in a called procedure's markstack that points to the markstack of the calling procedure.

dynamic variable a variable explicitly allocated by the program. Dynamic variables are allocated on the heap. (Contrast with **automatic variable**).

enhanced indirect addressing an addressing method used to extend the Apple III memory addressing beyond 64K bytes.

evaluation stack a data structure located on the user stack page. Used to pass parameters, to return function values, and as an operand source for many P-machine instructions. The evaluation stack grows downward.

execution time the period of time during which a program is running.

EXTERNAL function a declaration of a separate function. The declaration occurs in the calling procedure, and the actual native code occurs in a separate function.

EXTERNAL procedure a declaration of a separate procedure. The declaration occurs in the calling procedure, and the actual native code occurs in a separate procedure.

extra code space the portion of memory that is not occupied by SOS, the interpreter, BIOS, device drivers, the program stack-heap, or graphics space. Code is automatically loaded in extra code space if any is available.

function a procedure that returns a value.

global an entity accessible to all procedures within the scope of the procedure that declares it. See the *Apple III Pascal Programmer's Manual* for a discussion of scope.

global procedure a procedure of lexical level Ø.

heap part of the Apple III memory space used by the Pascal operating system to store dynamic variables. The heap grows toward the stack.

high byte bits 8 to 15 of a word.

host program a program in which other units or assembly procedures may be used.

host program global data area the data area in the host program's global activation record that holds variables declared at the outermost lexical level of the host program (level −1 or Ø).

IMPLEMENTATION the portion of a unit following the INTERFACE. The IMPLEMENTATION contains declarations of private constants, types, and variables, private procedures, and functions, and the actual P-code of the procedures and functions declared in the INTERFACE.

interface text the portion of a code segment that contains the ASCII text of the INTERFACE in the source text of a unit.

INTERFACE the portion of a unit following the unit heading. The INTERFACE contains declarations of constants, types, variables, procedures, and functions that are made available to programs that USE the unit.

intrinsic unit a unit whose code remains in its library codefile until the host program is executed. The Linker is not needed for intrinsic units; they are "prelinked."

interpreter-relative relocation table a table of addresses, within an assembly-language procedure, each address to be relocated relative to a table within the interpreter.

IPC Interpreter Program Counter. A pointer on zero page that contains the address of the next instruction to be executed in the currently executing procedure. See **XIPC**.

JTAB Jump TABle pointer. A 16-bit pointer on zero page that points to the highest word of the attribute table of the currently executing procedure. See **XJTAB**.

jump table a section of self-relative pointers to addresses within the procedure code used by jump instructions. Jump tables are located at the bottom of attribute tables.

KP program stacK Pointer. A 16-bit pointer on zero page that points to the bank-pair address of the current top of the program stack. See **XKP**.

label an identifier.

lexical level the level of procedure nesting within a program. The user program is lexical level Ø; a procedure nested n levels deep within the user program has lexical level n.

LIBMAP utility program a Pascal program that creates a Map textfile of Linker information, interface text, procedures, and functions for each segment in a library.

Librarian a Pascal system program used to combine separately compiled or assembled codefiles into a single codefile or library file.

library file a codefile containing intrinsic units, regular units, and/or external assembly-language procedures that can be used by a host program.

library name file an ASCII file that contains one to five names of library files used by a host program.

linked file a codefile that results from linking a host program segment with its referenced units and separate procedures and functions.

Linker a system program used to incorporate separately compiled or assembled procedures into a host program.

Linker information the portion of a code segment that enables the Linker to resolve references and definitions of identifiers between separately compiled or assembled code.

linker information type a record within Linker information that indicates the specific kind of reference or declaration that the Linker must resolve.

local entity an entity accessible only to the specific procedure within which it was declared.

LONGINTIO a standard library unit that provides long integer arithmetic operations and the built-in STR function.

low byte bits 0 to 7 of a word.

machine type the kind of microprocessor, for example, 6502.

main procedure the lowest level procedure in a segment.

markstack the lower part of an activation record that contains addressing context information and information on a calling procedure's environment.

MP Markstack Pointer. A 16-bit pointer on zero page that holds the address of the MSSTAT field in the topmost markstack on the program stack. See **XMP**.

native code assembled code for a microprocessor.

NEXTSEG Compiler option a Compiler option that allows you to specify the segment number of the next regular unit, SEGMENT procedure, or SEGMENT function encountered by the Compiler.

NP New Pointer. 16-bit pointer on zero page that points to the local bank-pair address of the current top of the heap (one byte above the last byte in use). See **XNP**.

operand a single value, such as a constant, variable, reference, or function call.

page a unit of storage comprising two blocks, or 1024 contiguous bytes.

PASCALIO a standard library unit that holds the SEEK, WRITE, WRITELN, READ, and READLN procedures.

POINTERLIST a list of pointers in Linker information, each of which points to a location within the code segment where there is a reference to a variable, identifier, or constant that must be fixed up by the Linker.

private an entity held in the global data area, but not accessible to the user program.

procedure a section of procedure code with an accompanying attribute table. The term *procedure* is used to refer to the main program, any procedure, or any function.

procedure code a sequence of native code or P-code instructions.

procedure dictionary the upper section of a segment's code part, containing a list of pointers to the procedures in the code part.

procedure number a number used to refer to a specific procedure.

procedure-relative relocation table a table of addresses, within an assembly-language procedure, each to be relocated relative to the lowest address in the procedure.

program library a library file with intrinsic units only.

program stack a portion of memory used to store automatic variables, bookkeeping information about procedure and function calls, and code, if there is no available extra code space.

psuedo-code or **P-code** the compiled form of a Pascal program. Psuedo-code is a machine-independent intermediate code that is interpreted by a specific machine-dependent interpreter.

pseudo-machine or **P-machine** a software-emulated machine that executes P-code as its native code. The P-machine has an evaluation stack, several registers, and a user memory.

psuedo-register a P-machine pointer composed of one word on zero page, and an X-byte on X-page (except for the SP register).

regular unit a unit whose code is separately compiled and combined with the host program's codefile by the Linker.

relocation table a sequence of records that contain information necessary to relocate any relocatable addresses, within a native-code procedure, whenever the segment containing the procedure is loaded into memory.

SEG SEGment pointer. A 16-bit pointer on zero page that holds the local bank-pair address of the highest word of the procedure dictionary of the segment to which the currently executing procedure belongs. See **XSEG**.

segment a section of a Pascal program that can be swapped in or out of memory as required for operation.

segment dictionary block Ø of a codefile that contains information needed by the Pascal system to load and execute the segments in the codefile.

SEGMENT function a function that comprises its own unique segment. The code of SEGMENT functions is not loaded into memory until the function is called; as soon as it terminates, the space occupied by the code can be used for something else.

segment number a unique number assigned to each segment. Used as an index into the segment table.

SEGMENT procedure a procedure that comprises its own unique segment. The code of SEGMENT procedures is not loaded into memory until the procedure is called; as soon as it terminates, the space occupied by the code can be used for something else.

segment-relative relocation table a table of addresses, within an assembly-language procedure, each to be relocated relative to the lowest address in the segment.

segment table a section of the higher addresses of SYSCOM that comprise a list containing information needed by the P-machine to read code segments into memory or to allocate space for data segments.

self-relative pointer a pointer that points to an address, relative to the location of itself. To find the address referred to by a self-relative pointer, subtract the pointer from the address of its location.

separate function a separately-compiled assembly-language function in a library. Separate functions must be defined as external functions in the calling procedure.

separate procedure a separately-compiled assembly-language procedure in a library. Separate procedures must be defined as external procedures in the calling procedure.

slot one of the 16 entries in a segment dictionary. There is one slot for each segment in the codefile.

SP evaluation Stack Pointer. An 8-bit pointer to the current *top* of the evaluation stack. It is actually the Apple III hardware stack pointer.

stack see program stack.

stack/heap space a portion of memory used exclusively by the program stack and heap.

static chain a series of static links. A static chain describes the lexical *nesting* levels of a procedure.

static link a pointer in a called procedure's markstack that points to the markstack of the procedure's lexical parent.

STRP STRing Pointer. A 16-bit pointer on zero page that points to the bank-pair address of the top of the linked list of packed arrays of characters and strings on the stack. See **XSTRP**.

SYSCOM a section of memory on the stack used by the operating system and the P-machine to exchange information.

SYSTEM.LIBRARY file a library file that contains a group of separately-compiled Pascal system procedures and functions.

textfile a file containing human-readable text, such as a source program.

tos the operand on the top of the evaluation stack.

unit a collection of procedures that are separately compiled into libraries and then invoked as modular components of a user program.

unit info the last ten characters in an interface text, necessary for the Compiler to compile a code segment that uses the interface text.

unlinked file a file that has not been linked with its calling procedure and procedures that it calls.

user memory the portion of memory not occupied by SOS, the interpreter, BIOS, and device drivers.

user program the main procedure of segment number 1.

user program global data area an area of memory that holds variables declared at the outermost lexical level of the user program (level 0).

word 16 bits or two bytes, of which the lower, even-address byte is least significant on the 6502.

word-aligned an instruction or structure starting at an even byte (see **byte-aligned**).

XBASE BASE procedure pointer. A pointer on X-page that contains the number of the bank-pair for the MSSTAT field of the activation record of the most recently invoked base procedure. See **BASE**.

X-byte a byte on X-page, used to facilitate enhanced indirect addressing. Also termed the eXtension-byte.

XIPC Interpreter Program Counter. A pointer on X-page that contains the number of the bank pair for the address of the next instruction to be executed in the currently executing procedure. See **IPC**.

XJTAB Jump TABle pointer. A pointer on X-page that contains the number of the bank pair for the highest word of the attribute table in the procedure code of the currently executing procedure. See **JTAB**.

XKP program stacK Pointer. A pointer on X-page that contains the number of the bank-pair for the current top of the program stack. See **KP**.

XMP Markstack Pointer. A pointer on X-page that contains the number of the bank-pair for the MSSTAT field in the topmost markstack on the program stack. See **MP**.

XNP New Pointer. A pointer on X-page that contains the number of the bank-pair for the current top of the heap (one byte above the last byte in use). See **NP**.

X-page locations $1600 through $16FF. Also called the extension page. The X-bytes of the system psuedo-registers reside here.

XSEG SEGment pointer. A pointer on X-page that contains the number of the bank-pair for the highest word of the procedure dictionary of the segment to which the currently executing procedure belongs. See **SEG**.

XSTRP STRing Pointer. A pointer on X-page that contains the number of the bank-pair for the top of the linked list of packed arrays of characters and strings on the stack. See **STRP**.

zero page locations $00 through $FF. The system psuedo-registers reside here.

Index

B

G

N

[redacted]